あのクルマの マテリアルを再現！？ 微分可能レンダラを用いた BRDFフィッティング

ポリフォニー・デジタル
内村 創

自己紹介

- 画像処理
- 色彩工学
- 質感計測

自己紹介です。内村はじめといいます。
ゲーム内アセットの画像圧縮だとか、
色彩工学だとか、質感計測だとか
そういった分野を担当しています。

Agenda

- Introduction
- ゲームにおけるマテリアルとは
- 現実世界の素材の何を計測するのか
- 今までの測色復元システムについて
- 新しい測色復元システムについて
  - 微分可能レンダリングについて
  - 実際の実装と結果
- 課題と将来

POLYPHONY™ DIGITAL

Introduction

グランツーリスモは実在メーカーの実在車両を精密に再現することを特徴とするレースゲームです。

継続的なアップデートによって、収録車種の幅も広くなってきており、再現する車両の塗装の種類も多岐に渡ります。

実車の塗装は多種多様

実在車両の塗装は、非常に多種多様です。

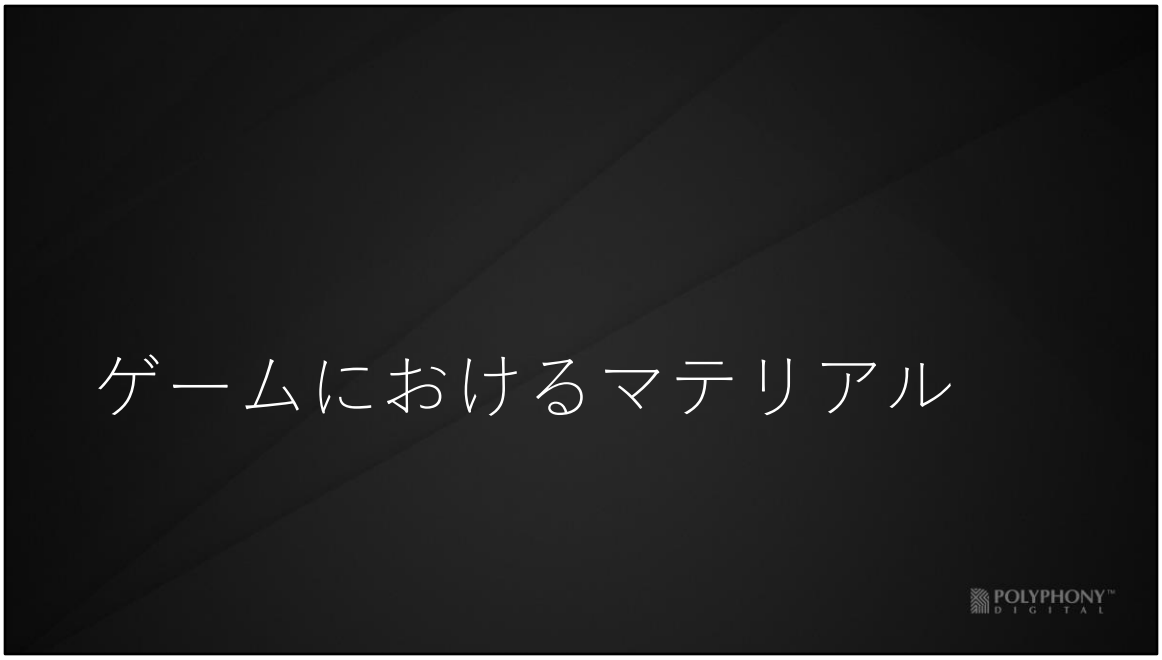
例えば東京オートサロンなどに行けば、信じられないようなシェーダのアサインされたクルマを何台も見ることができます。

現実世界のマテリアル設定は本当に多種多様です。

この模様は、ボンネットの表面の金属にマイクロスクラッチを磨き入れた上に色付きのクリアコートを塗ったものです。

これらの塗装の写真を見るだけでは、シェーダをどういったパラメータに設定すればいいか、パッと想像することは難しいのではないでしょうか。

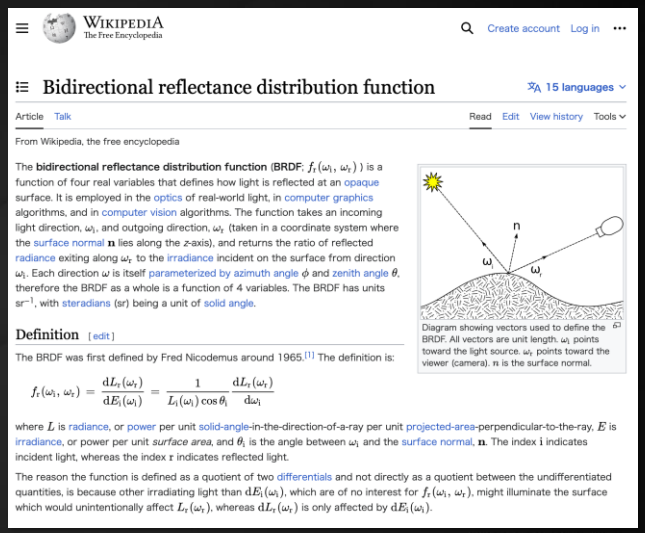
グランツーリスモシリーズは毎回非常に多数の車種を収録しています。
それぞれのクルマごとに複数の色設定がありますので、
その全てをアーティストによって手で調整するのは現実的ではありません。
弊社には以前から、計測データからゲーム内マテリアルのパラメータを求めるシステムへの需要がありました。



まず、ゲームにおけるマテリアルを考えましょう

BRDF

双方向反射率分布関数


WIKIPEDIA
The Free Encyclopedia
Create account
Log in
Bidirectional reflectance distribution function
15 languages
Article
Talk
Read
Edit
View history
Tools
From Wikipedia, the free encyclopedia
The bidirectional reflectance distribution function (BRDF; $f_r(\omega_i, \omega_r)$) is a function of four real variables that defines how light is reflected at an opaque surface. It is employed in the optics of real-world light, in computer graphics algorithms, and in computer vision algorithms. The function takes an incoming light direction, ω_i, and outgoing direction, ω_r (taken in a coordinate system where the surface normal $\mathbf{n}$ lies along the z-axis), and returns the ratio of reflected radiance exiting along ω_r to the irradiance incident on the surface from direction ω_i. Each direction ω is itself parameterized by azimuth angle ϕ and zenith angle θ, therefore the BRDF as a whole is a function of 4 variables. The BRDF has units sr^{-1}, with steradians (sr) being a unit of solid angle.
n
ω_i
ω_r
Diagram showing vectors used to define the BRDF. All vectors are unit length. ω_i points toward the light source. ω_r points toward the viewer (camera). n is the surface normal.
Definition [edit]
The BRDF was first defined by Fred Nicodemus around 1965.[1] The definition is:
$$f_r(\omega_i, \omega_r) = \frac{\mathrm{d}L_r(\omega_r)}{\mathrm{d}E_i(\omega_i)} = \frac{1}{L_i(\omega_i)\cos\theta_i}\frac{\mathrm{d}L_r(\omega_r)}{\mathrm{d}\omega_i}$$
where L is radiance, or power per unit solid-angle-in-the-direction-of-a-ray per unit projected-area-perpendicular-to-the-ray, E is irradiance, or power per unit *surface area*, and θ_i is the angle between ω_i and the surface normal, $\mathbf{n}$. The index i indicates incident light, whereas the index r indicates reflected light.
The reason the function is defined as a quotient of two differentials and not directly as a quotient between the undifferentiated quantities, is because other irradiating light than $\mathrm{d}E_i(\omega_i)$, which are of no interest for $f_r(\omega_i, \omega_r)$, might illuminate the surface which would unintentionally affect $L_r(\omega_r)$, whereas $\mathrm{d}L_r(\omega_r)$ is only affected by $\mathrm{d}E_i(\omega_i)$.

[Wikipedia:BRDF]

POLYPHONY DIGITAL

ある質感を再現しようと思った場合、最も重要なのは、BRDFです。
BRDFは日本語では
双方向反射率分布関数
と呼ばれます。

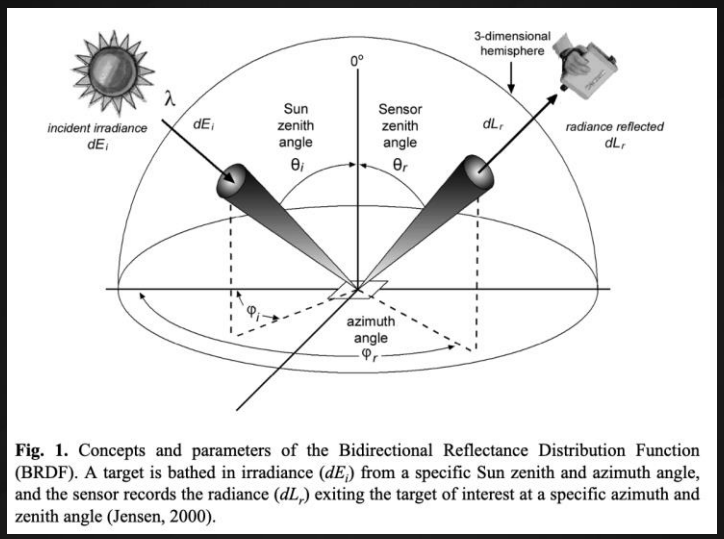


Fig. 1. Concepts and parameters of the Bidirectional Reflectance Distribution Function (BRDF). A target is bathed in irradiance (dE_i) from a specific Sun zenith and azimuth angle, and the sensor records the radiance (dL_r) exiting the target of interest at a specific azimuth and zenith angle (Jensen, 2000).

[Schill2013]

POLYPHONY™ DIGITAL

BRDFは、簡単にいってしまえば、入射方向と、出射方向を引数に取り、その反射率を返す関数です。

BRDF

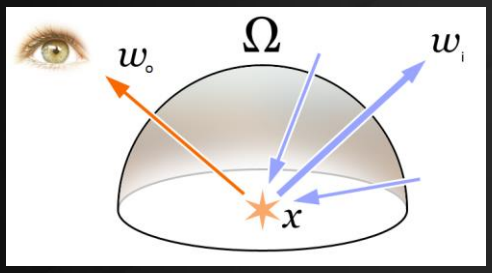
$$L_o(x, \overrightarrow{\omega_o}) = L_e(x, \overrightarrow{\omega_o}) + \int_\Omega f_r(x, \overrightarrow{\omega_i}, \overrightarrow{\omega_o}) L_i(x, \overrightarrow{\omega_i})(\overrightarrow{\omega_i} \cdot \vec{n}) d\overrightarrow{\omega_i}$$

POLYPHONY DIGITAL

BRDFは、レンダリング方程式でいうところの、この fr の項に該当します。

レンダリング方程式とは

$$L_o(x,\overrightarrow{\omega_o}) = L_e(x,\overrightarrow{\omega_o}) + \int_{\Omega} f_r(x,\overrightarrow{\omega_i},\overrightarrow{\omega_o}) L_i(x,\overrightarrow{\omega_i})(\overrightarrow{\omega_i}\cdot\overrightarrow{\omega_n}) d\overrightarrow{\omega_i}$$



[Wikipedia:Rendering]

x	位置
$\overrightarrow{\omega_o}$	出射ベクトル
$\overrightarrow{\omega_i}$	入射ベクトル
$\overrightarrow{\omega_n}$	法線ベクトル
$L_e(x,\overrightarrow{\omega_o})$	自己発光
$L_i(x,\overrightarrow{\omega_i})$	入射輝度
$f_r(x,\overrightarrow{\omega_i},\overrightarrow{\omega_o})$	BRDF

POLYPHONY DIGITAL

レンダリング方程式とは、ある点に入ってくる光が、目の方向にどうやって届くかを表した方程式です。

BRDF

$$f_r(x, \overrightarrow{\omega_i}, \overrightarrow{\omega_o}) = \frac{dL_r(x, \overrightarrow{\omega_o})}{dE_i(x, \overrightarrow{\omega_i})}$$

x	位置
$\overrightarrow{\omega_o}$	出射ベクトル
$\overrightarrow{\omega_i}$	入射ベクトル
$L_r(x, \vec{\omega})$	反射放射輝度
$dE_i(x, \overrightarrow{\omega_i})$	微小放射照度

POLYPHONY™ DIGITAL

BRDFは、式で書くとこのような定義になります。

BRDF

$$f_r(x, \overrightarrow{\omega_i}, \overrightarrow{\omega_o}) = \frac{dL_r(x, \overrightarrow{\omega_o})}{dE_i(x, \overrightarrow{\omega_i})}$$

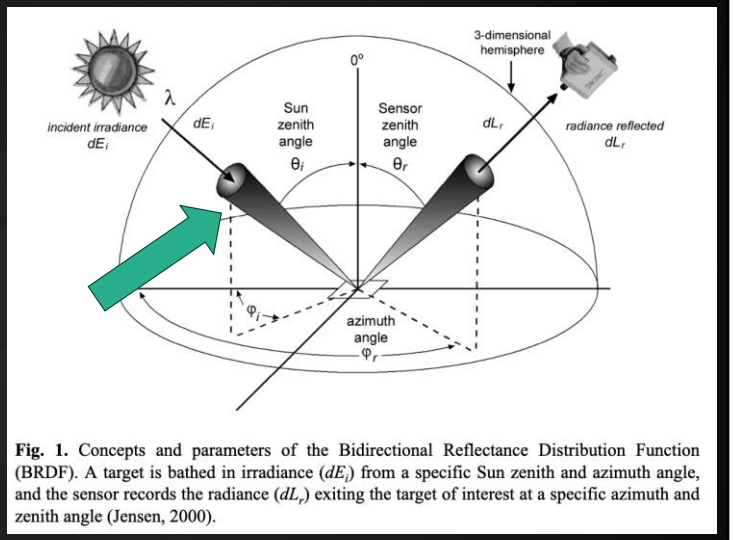


Fig. 1. Concepts and parameters of the Bidirectional Reflectance Distribution Function (BRDF). A target is bathed in irradiance (dE_i) from a specific Sun zenith and azimuth angle, and the sensor records the radiance (dL_r) exiting the target of interest at a specific azimuth and zenith angle (Jensen, 2000).

先ほどの図で言うところの、この入射方向のベクトルと

BRDF

$$f_r(x, \overrightarrow{\omega_i}, \overrightarrow{\omega_o}) = \frac{dL_r(x, \overrightarrow{\omega_o})}{dE_i(x, \overrightarrow{\omega_i})}$$

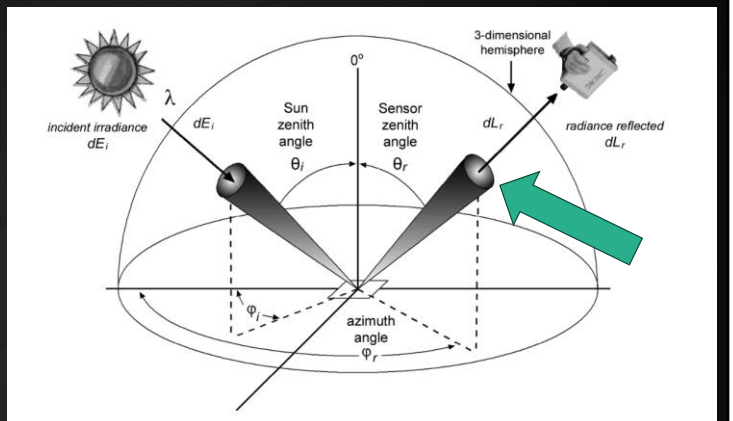


Fig. 1. Concepts and parameters of the Bidirectional Reflectance Distribution Function (BRDF). A target is bathed in irradiance (dE_i) from a specific Sun zenith and azimuth angle, and the sensor records the radiance (dL_r) exiting the target of interest at a specific azimuth and zenith angle (Jensen, 2000).

出射方向のベクトルと

BRDF

$$f_r(x, \overrightarrow{\omega_i}, \overrightarrow{\omega_o}) = \frac{dL_r(x, \overrightarrow{\omega_o})}{dE_i(x, \overrightarrow{\omega_i})}$$

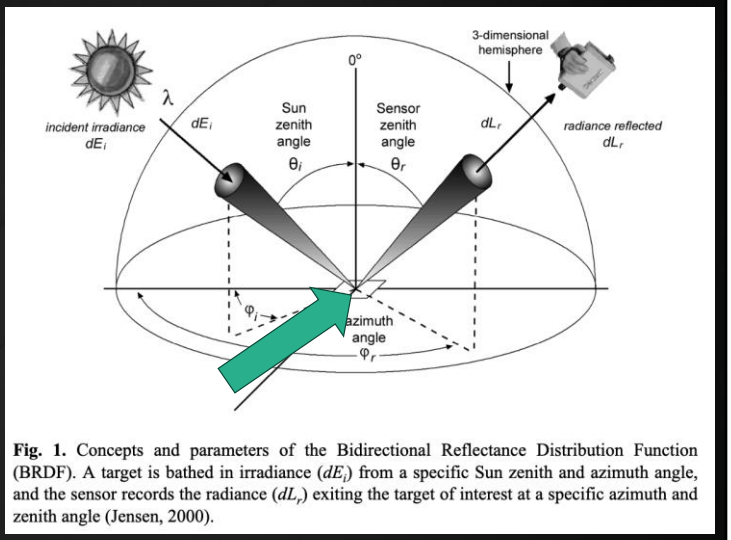


Fig. 1. Concepts and parameters of the Bidirectional Reflectance Distribution Function (BRDF). A target is bathed in irradiance (dE_i) from a specific Sun zenith and azimuth angle, and the sensor records the radiance (dL_r) exiting the target of interest at a specific azimuth and zenith angle (Jensen, 2000).

位置を引数に取る関数です。

BRDF

$$f_r(x, \overrightarrow{\omega_i}, \overrightarrow{\omega_o}) = \frac{dL_r(x, \overrightarrow{\omega_o})}{dE_i(x, \overrightarrow{\omega_i})}$$

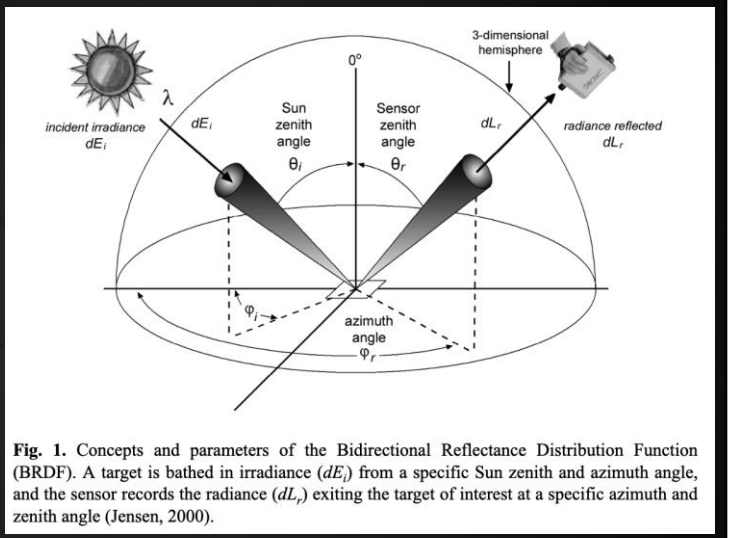


Fig. 1. Concepts and parameters of the Bidirectional Reflectance Distribution Function (BRDF). A target is bathed in irradiance (dE_i) from a specific Sun zenith and azimuth angle, and the sensor records the radiance (dL_r) exiting the target of interest at a specific azimuth and zenith angle (Jensen, 2000).

ここでは、簡単のため、同じマテリアルはどの位置でも同じBRDFであると仮定し、位置xについては考えないことにします。

塗装とは？

- 下塗り＋クリアコート
- 間に金属フレーク

4.3　塗膜設計

ねらいの光学特性を実現するため，塗膜内の光の経路に沿って各塗膜層の機能を設計した。まず，「シャープな赤の波長分布」を透過層に分担させた。透過層は，赤以外の波長の光を効果的に吸収させる機能に特化させる。そして，透過層の下層に「金属の反射特性」と「ハイライトとシェードの光量差」を分担させる。具体的にはマシーングレープレミアムメタリックの塗膜構造である反射層と吸収層の組み合わせで実現できると考えた。

実際に透過層，反射層，吸収層の3層構成で，既存原材料を組み合わせて試作品（Fig.10）を作った結果，ねらいの光学特性への実現性と課題が明らかになった。

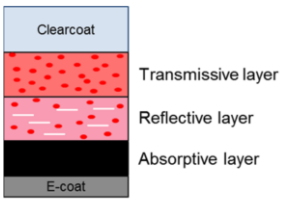
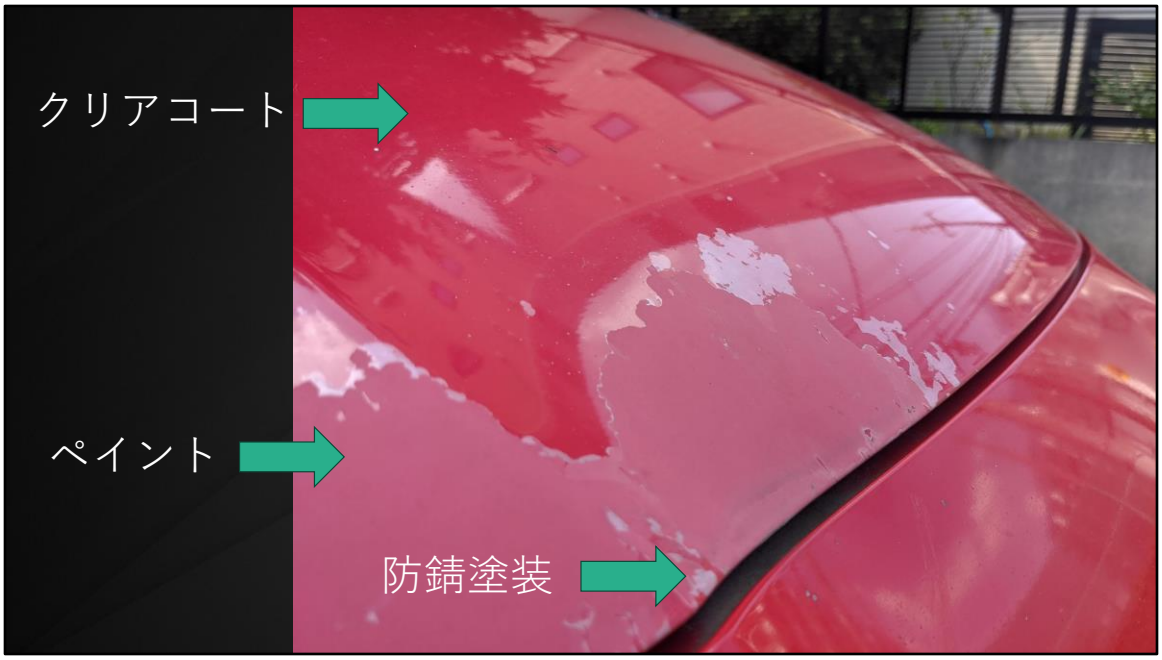


Fig. 10　Film Structure

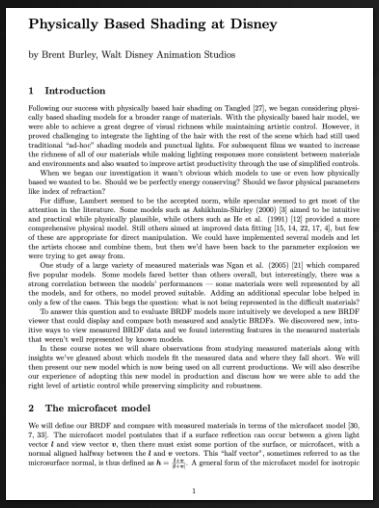
[MazdaTR2017]

さて、実際の車両の塗装は、このような多層構造になっていることが知られています。



先日道端で見つけて、あまりにちょうどいいリファレンスだったのでつい写真をたくさん撮ってしまったのですが、
実物の車の塗装が、説明通り、複数の層になっていることがわかるかとおもいます。

Principled BRDF



Physically Based Shading at Disney

by Brent Burley, Walt Disney Animation Studios

1 Introduction

Following our success with physically based hair shading on Tangled [27], we began considering physically based shading models for a broader range of materials. With the physically based hair model, we were able to achieve a great degree of visual richness while maintaining artistic control. However, it proved challenging to integrate the lighting of the hair with the rest of the scene which had still used traditional "ad-hoc" shading models and punctual lights. For subsequent films we wanted to increase the richness of all of our materials while making lighting responses more consistent between materials and environments and also wanted to improve artist productivity through the use of simplified controls.

When we began our investigation it wasn't obvious which models to use or even how physically based we wanted to be. Should we be perfectly energy conserving? Should we favor physical parameters like index of refraction?

For diffuse, Lambert seemed to be the accepted norm, while specular seemed to get most of the attention in the literature. Some models such as Ashikhmin-Shirley (2000) [3] aimed to be intuitive and practical while physically plausible, while others such as He et al. (1991) [12] provided a more comprehensive physical model. Still others aimed at improved data fitting [15, 14, 22, 17, 4], but few of these are appropriate for direct manipulation. We could have implemented several models and let the artists choose and combine them, but then we'd have been back to the parameter explosion we were trying to get away from.

One study of a large variety of measured materials was Ngan et al. (2005) [21] which compared five popular models. Some models fared better than others overall, but interestingly, there was a strong correlation between the models' performances — some materials were well represented by all the models, and for others, no model proved suitable. Adding an additional specular lobe helped in only a few of the cases. This begs the question: what is not being represented in the difficult materials?

To answer this question and to evaluate BRDF models more intuitively we developed a new BRDF viewer that could display and compare both measured and analytic BRDFs. We discovered new, intuitive ways to view measured BRDF data and we found interesting features in the measured materials that weren't well represented by known models.

In these course notes we will share observations from studying measured materials along with insights we've gleaned about which models fit the measured data and where they fall short. We will then present our new model which is now being used on all current productions. We will also describe our experience of adopting this new model in production and discuss how we were able to add the right level of artistic control while preserving simplicity and robustness.

2 The microfacet model

We will define our BRDF and compare with measured materials in terms of the microfacet model [30, 7, 33]. The microfacet model postulates that if a surface reflection can occur between a given light vector $\mathbf{l}$ and view vector $\mathbf{v}$, then there must exist some portion of the surface, or microfacet, with a normal aligned halfway between the $\mathbf{l}$ and $\mathbf{v}$ vectors. This "half vector", sometimes referred to as the microsurface normal, is thus defined as $\mathbf{h} = \frac{\mathbf{l}+\mathbf{v}}{\|\mathbf{l}+\mathbf{v}\|}$. A general form of the microfacet model for isotropic

1

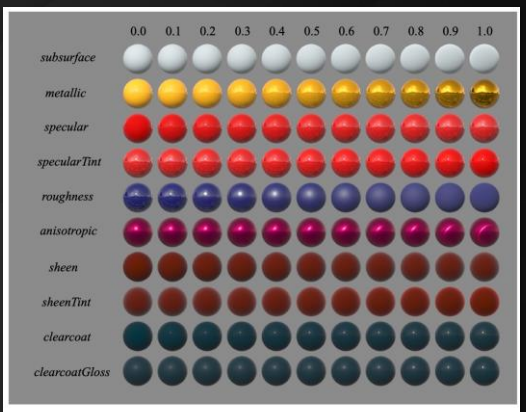


Figure 16: Examples of the effect of our BRDF parameters. Each parameter is varied across the row from zero to one with the other parameters held constant.

[Burley2012]

この多層構造の塗装を、少ないパラメータでそれなりに説得力をもってレンダリングできるBRDFモデルとして、
Principled BRDF が知られています。フォトリアルを目指すゲームタイトルは、その多くがPrincipled BRDFを用いています。
グランツーリスモシリーズでも、独自の改良を加えたPrincipled BRDFベースのマテリアルを利用してきました。

Principled BRDF

$$f_r(\omega_i, \omega_o) =$$

$$\left(\begin{array}{c} f_{diffuse}(\omega_i, \omega_o) * \mathrm{s} + \\ f_{subsurface}(\omega_i, \omega_o) * (1 - \mathrm{s}) + \\ f_{sheen}(\omega_i, \omega_o) \end{array} \right) * (1 - m)$$

$$+ f_{specular}(\omega_i, \omega_o)$$
$$+ f_{clearcoat}(\omega_i, \omega_o)$$

POLYPHONY™ DIGITAL

Principled BRDF は、このような、幾つものBRDFをブレンドし加算した構造になっています。

Principled BRDF

$$f_r(\omega_i, \omega_o) =$$

$$\left(\begin{array}{c} f_{diffuse}(\omega_i, \omega_o) * \mathrm{s} + \\ f_{subsurface}(\omega_i, \omega_o) * (1 - \mathrm{s}) + \\ f_{sheen}(\omega_i, \omega_o) \end{array} \right) * (1 - m)$$

$$+ f_{specular}(\omega_i, \omega_o)$$
$$+ f_{clearcoat}(\omega_i, \omega_o)$$

[Blender:Princpled]

POLYPHONY DIGITAL

それぞれ、ディフューズ

Principled BRDF

$$f_r(\omega_i, \omega_o) =$$

$$\left(\begin{array}{l} f_{diffuse}(\omega_i, \omega_o) * \mathrm{s} + \\ f_{subsurface}(\omega_i, \omega_o) * (1 - \mathrm{s}) + \\ f_{sheen}(\omega_i, \omega_o) \end{array} \right) * (1 - m)$$

$$+ f_{specular}(\omega_i, \omega_o)$$
$$+ f_{clearcoat}(\omega_i, \omega_o)$$

POLYPHONY DIGITAL

subsurface

Principled BRDF

$$f_r(\omega_i, \omega_o) =$$

$$\left(\begin{array}{c} f_{diffuse}(\omega_i, \omega_o) * \mathrm{s} + \\ f_{subsurface}(\omega_i, \omega_o) * (1 - \mathrm{s}) + \\ f_{sheen}(\omega_i, \omega_o) \end{array} \right) * (1 - m)$$

$$+ f_{specular}(\omega_i, \omega_o)$$
$$+ f_{clearcoat}(\omega_i, \omega_o)$$

POLYPHONY DIGITAL

布などの特徴である、sheen

Principled BRDF

$$f_r(\omega_i, \omega_o) =$$

$$\left(\begin{array}{c} f_{diffuse}(\omega_i, \omega_o) * \mathrm{s} + \\ f_{subsurface}(\omega_i, \omega_o) * (1 - \mathrm{s}) + \\ f_{sheen}(\omega_i, \omega_o) \end{array} \right) * (1 - m)$$

$$+ f_{specular}(\omega_i, \omega_o)$$

$$+ f_{clearcoat}(\omega_i, \omega_o)$$

POLYPHONY DIGITAL

スペキュラ

Principled BRDF

$$f_r(\omega_i, \omega_o) =$$

$$\left(\begin{array}{c} f_{diffuse}(\omega_i, \omega_o) * \mathrm{s} + \\ f_{subsurface}(\omega_i, \omega_o) * (1 - \mathrm{s}) + \\ f_{sheen}(\omega_i, \omega_o) \end{array} \right) * (1 - m)$$

$$+ f_{specular}(\omega_i, \omega_o)$$
$$+ f_{clearcoat}(\omega_i, \omega_o)$$

POLYPHONY DIGITAL

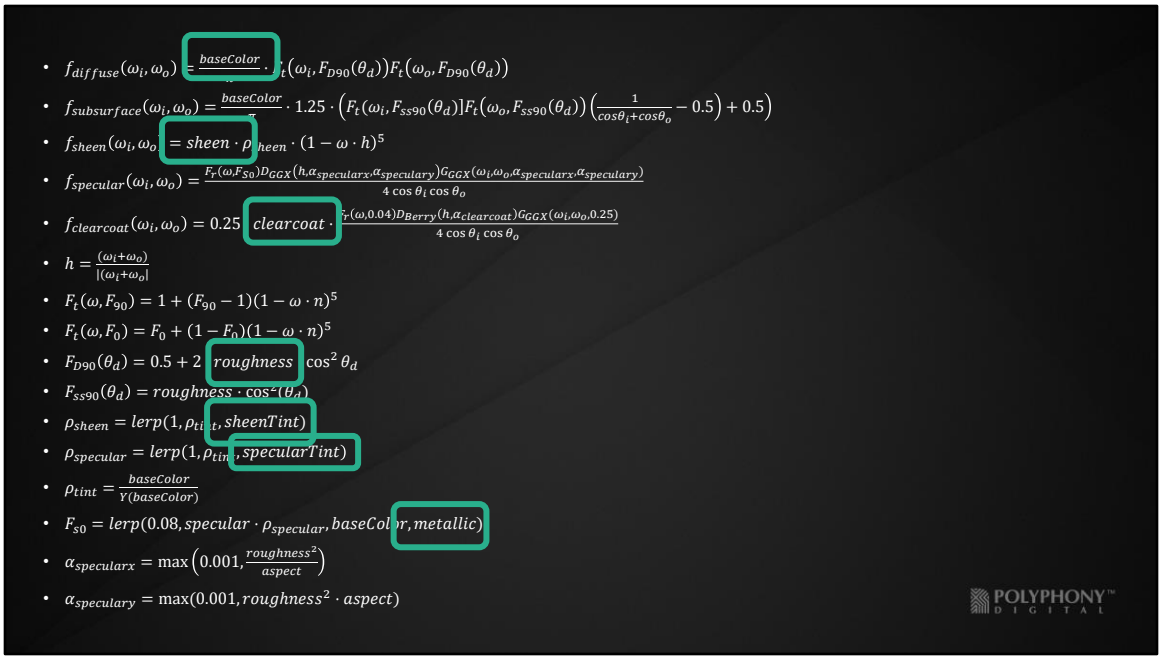
クリアコート

のようになっています。

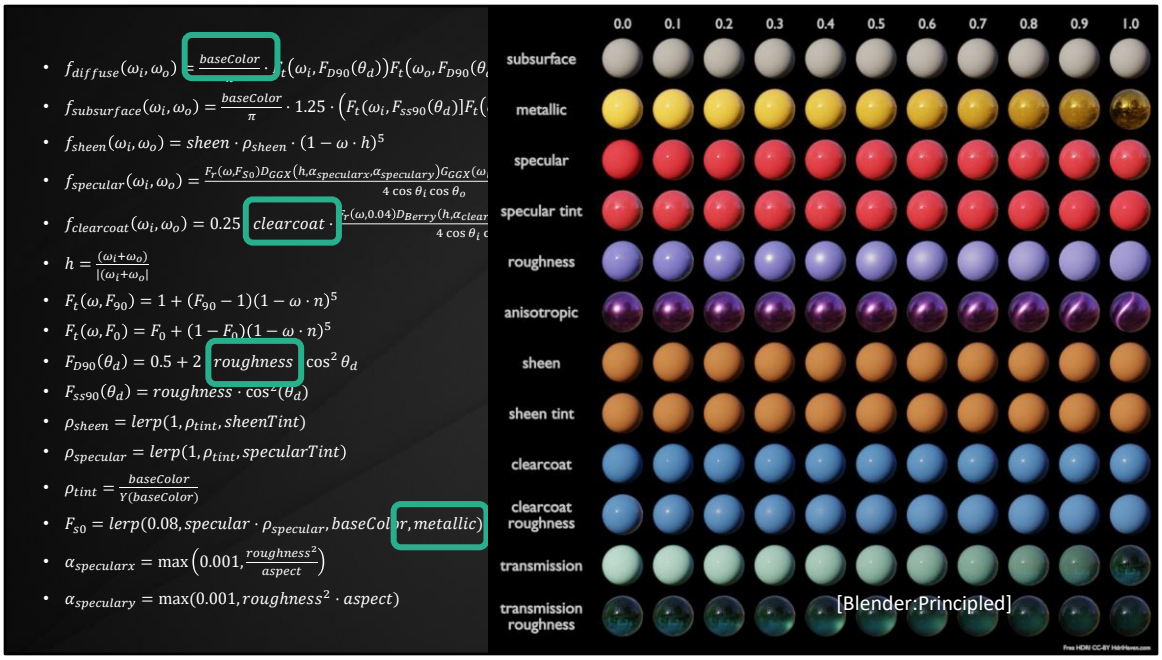
- $f_{diffuse}(\omega_i, \omega_o) = \frac{baseColor}{\pi} \cdot F_t(\omega_i, F_{D90}(\theta_d))F_t(\omega_o, F_{D90}(\theta_d))$
- $f_{subsurface}(\omega_i, \omega_o) = \frac{baseColor}{\pi} \cdot 1.25 \cdot \left(F_t(\omega_i, F_{ss90}(\theta_d)]F_t(\omega_o, F_{ss90}(\theta_d))\right)\left(\frac{1}{cos\theta_i + cos\theta_o} - 0.5\right) + 0.5\right)$
- $f_{sheen}(\omega_i, \omega_o) = sheen \cdot \rho_{sheen} \cdot (1 - \omega \cdot h)^5$
- $f_{specular}(\omega_i, \omega_o) = \frac{F_r(\omega, F_{S0})D_{GGX}(h, \alpha_{specularx}, \alpha_{speculary})G_{GGX}(\omega_i, \omega_o, \alpha_{specularx}, \alpha_{speculary})}{4 \cos\theta_i \cos\theta_o}$
- $f_{clearcoat}(\omega_i, \omega_o) = 0.25 \cdot clearcoat \cdot \frac{F_r(\omega, 0.04)D_{Berry}(h, \alpha_{clearcoat})G_{GGX}(\omega_i, \omega_o, 0.25)}{4 \cos\theta_i \cos\theta_o}$
- $h = \frac{(\omega_i + \omega_o)}{|(\omega_i + \omega_o|}$
- $F_t(\omega, F_{90}) = 1 + (F_{90} - 1)(1 - \omega \cdot n)^5$
- $F_t(\omega, F_0) = F_0 + (1 - F_0)(1 - \omega \cdot n)^5$
- $F_{D90}(\theta_d) = 0.5 + 2 \cdot roughness \cdot \cos^2\theta_d$
- $F_{ss90}(\theta_d) = roughness \cdot \cos^2(\theta_d)$
- $\rho_{sheen} = lerp(1, \rho_{tint}, sheenTint)$
- $\rho_{specular} = lerp(1, \rho_{tint}, specularTint)$
- $\rho_{tint} = \frac{baseColor}{Y(baseColor)}$
- $F_{s0} = lerp(0.08, specular \cdot \rho_{specular}, baseColor, metallic)$
- $\alpha_{specularx} = \max\left(0.001, \frac{roughness^2}{aspect}\right)$
- $\alpha_{specularly} = \max(0.001, roughness^2 \cdot aspect)$

POLYPHONY™ DIGITAL

それぞれの項はこのようになっていますが、今は詳細は追いません。



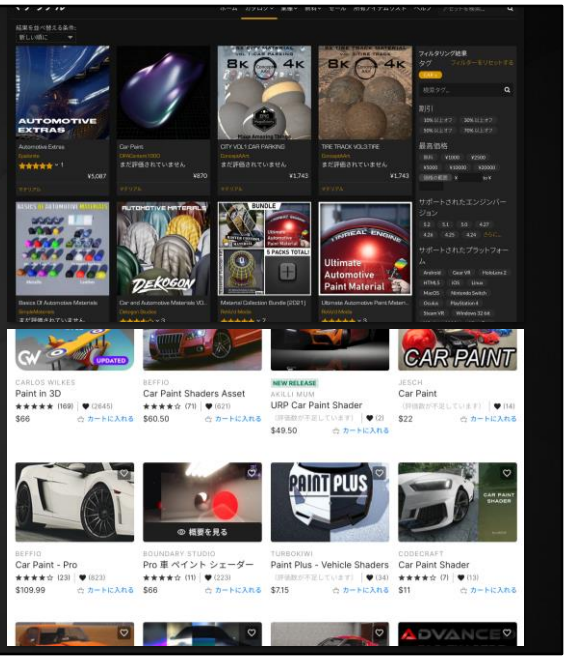
これらの項のうち、いくつかの変数が、このBRDFの調整パラメータになっています。



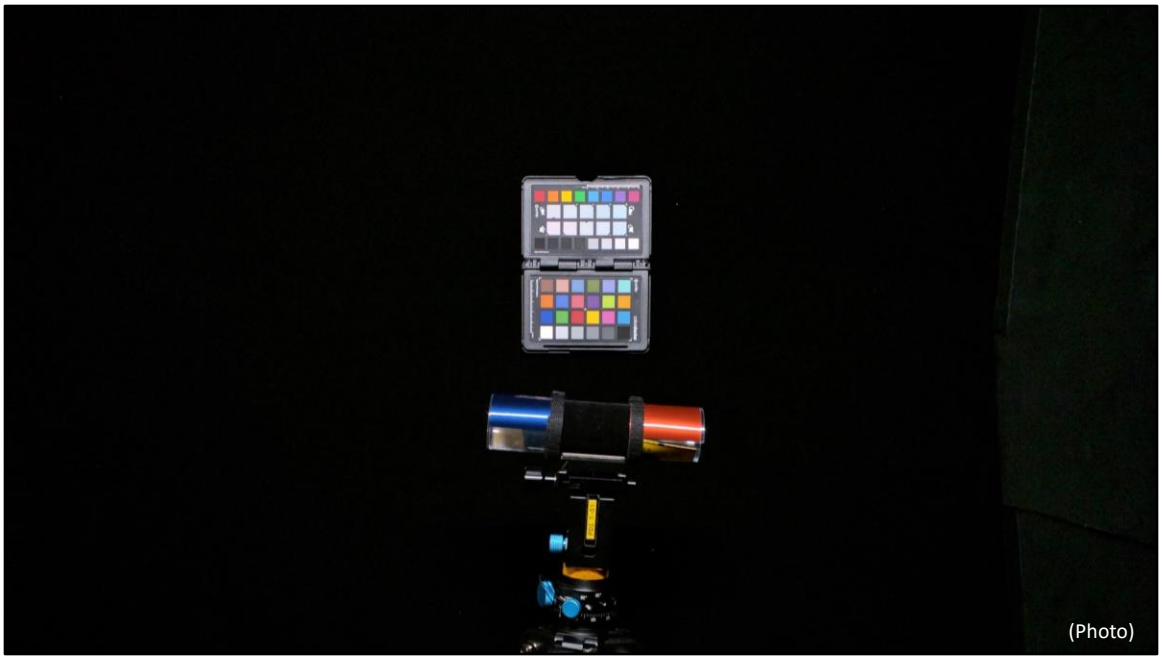
中でも、Basecolor, metallic, roughness, clearcoatの
四つのパラメータが、塗料などの再現には重要です。

車の塗装マテリアルは人気アセット

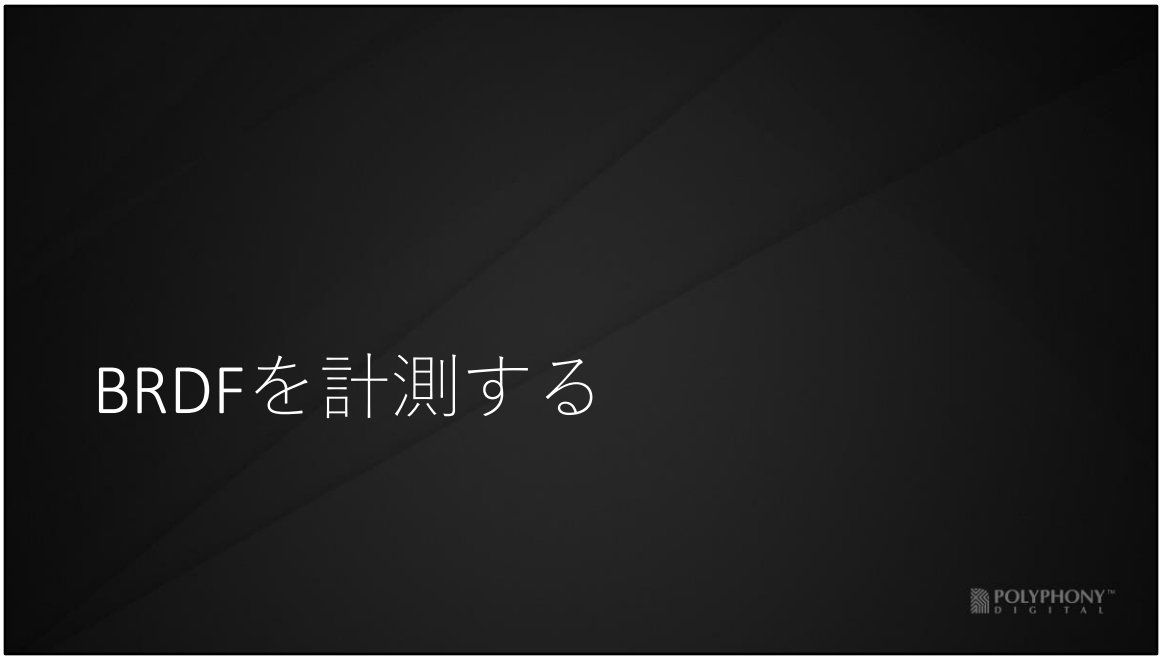
- ただし、手で調整するのは非常に面倒



車両塗装は、アンリアルエンジンやUnityでも人気のアセットですが、
それぞれのパラメータはあまり直感的でないことも多く、
調整に苦労された方も多いのではないでしょうか。
これらのアセットも、その多くが、PrincipledBRDFがベースになっています。



この写真は、暗幕を貼った閉鎖空間に、高演色な光源を設置した環境です。
こういった環境で実マテリアルを確認することができれば、マテリアルパラメータもある程度信頼のおける調整は可能です。
常にこの環境を用意することは、それほど容易ではありません。
そこで、安定してマテリアルのBRDFを計測する仕組みが重要になります。



BRDFを計測する

BRDF計測装置例

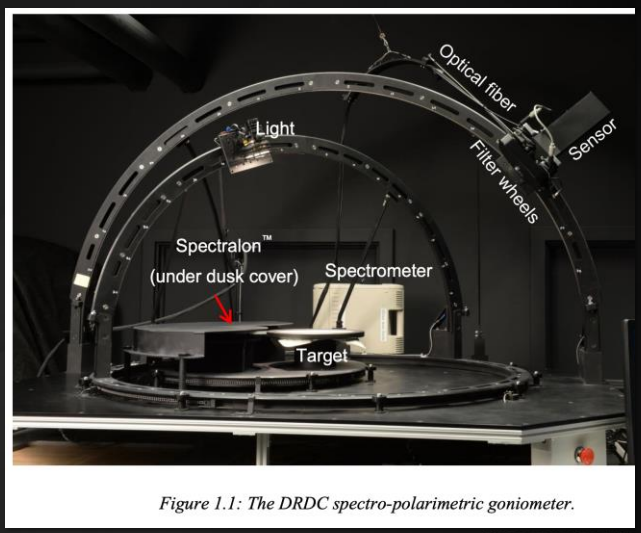
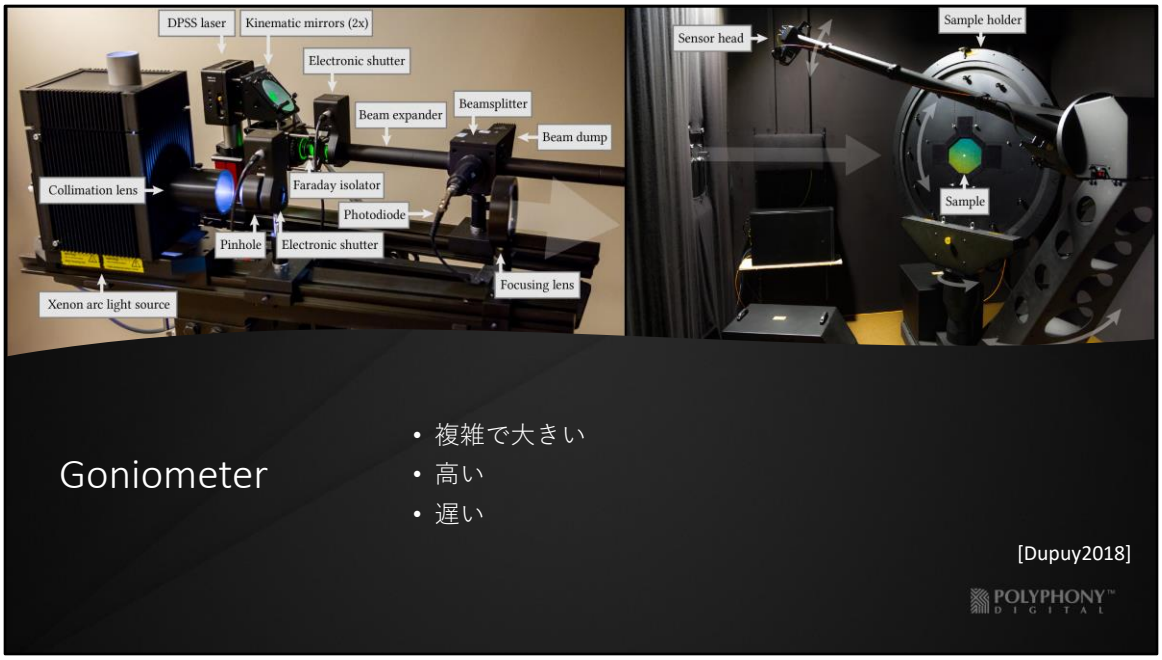


Figure 1.1: The DRDC spectro-polarimetric goniometer.

[DRDC2017]

BRDFの計測は、このようなゴニオメーターと呼ばれる装置を用いて行われるのが一般的です。



ゴニオメーターによるBRDF計測は非常に高精度ですが、いくつか問題点もあります。

それは、
複雑で大きいこと。
価格が高いこと。
計測が遅いこと。

です。

次元の呪い

$$f_r(\overrightarrow{\omega_i}, \overrightarrow{\omega_o})$$

$\overrightarrow{\omega_i}, \overrightarrow{\omega_o}$ともに2次元ずつなので
90*360*90*360 = 1,049,760,000

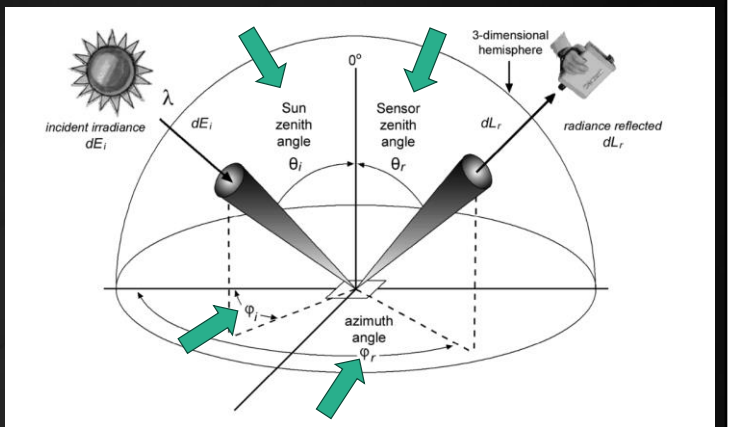


Fig. 1. Concepts and parameters of the Bidirectional Reflectance Distribution Function (BRDF). A target is bathed in irradiance (dE_i) from a specific Sun zenith and azimuth angle, and the sensor records the radiance (dL_r) exiting the target of interest at a specific azimuth and zenith angle (Jensen, 2000).

BRDFの定義から、入射ベクトル、出射ベクトルともに2次元ずつの自由度を持っていますので、
角度を1度刻みで計測したとしても、10億回サンプルする必要があるのです。

こうして多大な手間と時間をかけて計測したBRDFデータは、そのままではただの巨大な数値テーブルに過ぎません。
CGで直接利用できるようなマテリアルパラメータが出てくる計測器というのは、いまのところ存在しません。
粗さ標準片という、金属磨きの業界で利用されるスケールがあるのですが、目で見比べてもパラメータを得るのは非常に難しいです。
クリアコートやメタリックに至っては、そもそも現実世界には比較できる標準片もありません。
そこで、計測したBRDFデータに対して、BRDFの各パラメータをフィッティングするようなシステムが必要となります。

パラメータ推定

修士論文

二色性反射モデルに基づく反射の解析

高速反射成分分離

疎な形状データと密な法線マップを使用した
ロバスト反射パラメータ推定

Reflection Analysis Based on
Dichromatic Reflection Model

Fast Separation of Reflection Components

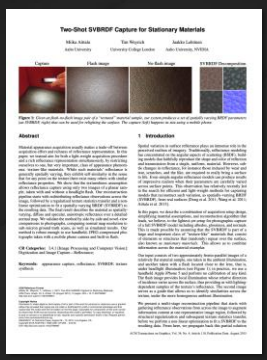
Robust Estimation of Reflection Parameters using
Rough Shape Data and Dense Normal Map

2008年2月4日提出
指導教官 池内克史 教授

東京大学
THE UNIVERSITY OF TOKYO

情報理工学系研究科 電子情報学専攻
66435 肥後 智昭

[Higo2008]


Two-Shot SVBRDF Capture for Stationary Materials

Miika Aittala
Aalto University

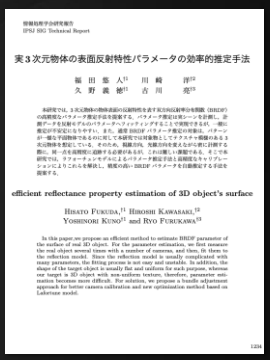
Tim Weyrich
University College London

Jaakko Lehtinen
Aalto University, NVIDIA

Abstract

1 Introduction

[Miika2015]


情報処理学会研究報告
IPSJ SIG Technical Report

実３次元物体の表面反射特性パラメータの効率的推定手法

福田 雅人[1] 川崎 洋[2]
久野 義徳[1] 吉川 恭[3]

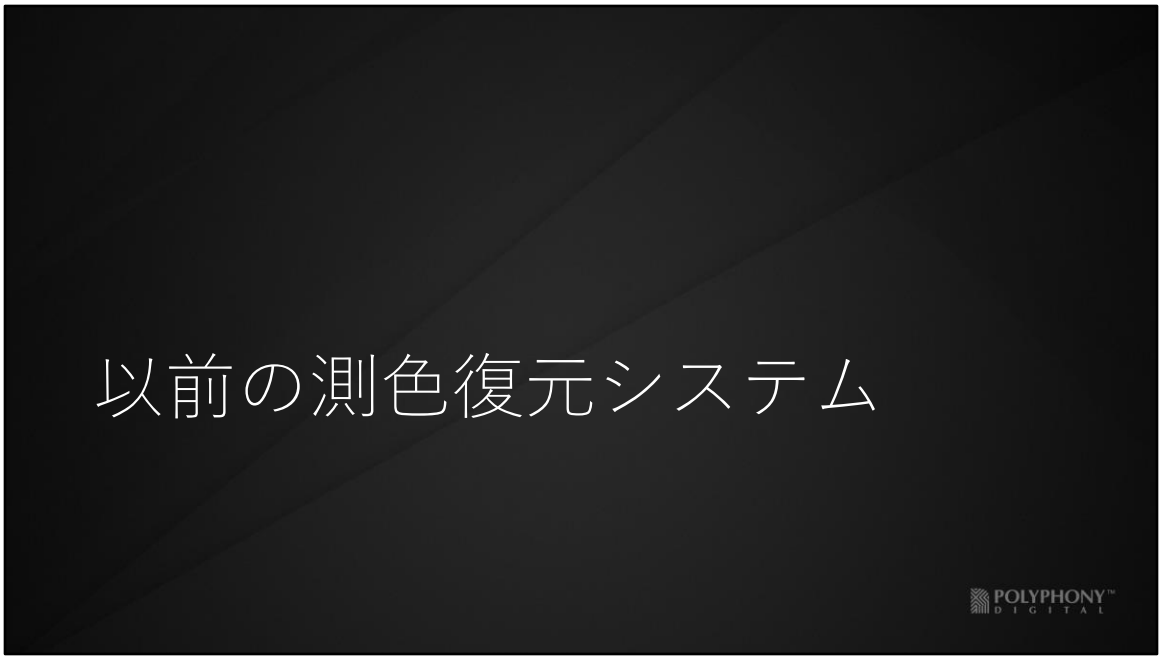
efficient reflectance property estimation of 3D object's surface

Hisato Fukuda,[1] Hiroshi Kawasaki,[2]
Yoshinori Kuno[1] and Ryo Furukawa[3]

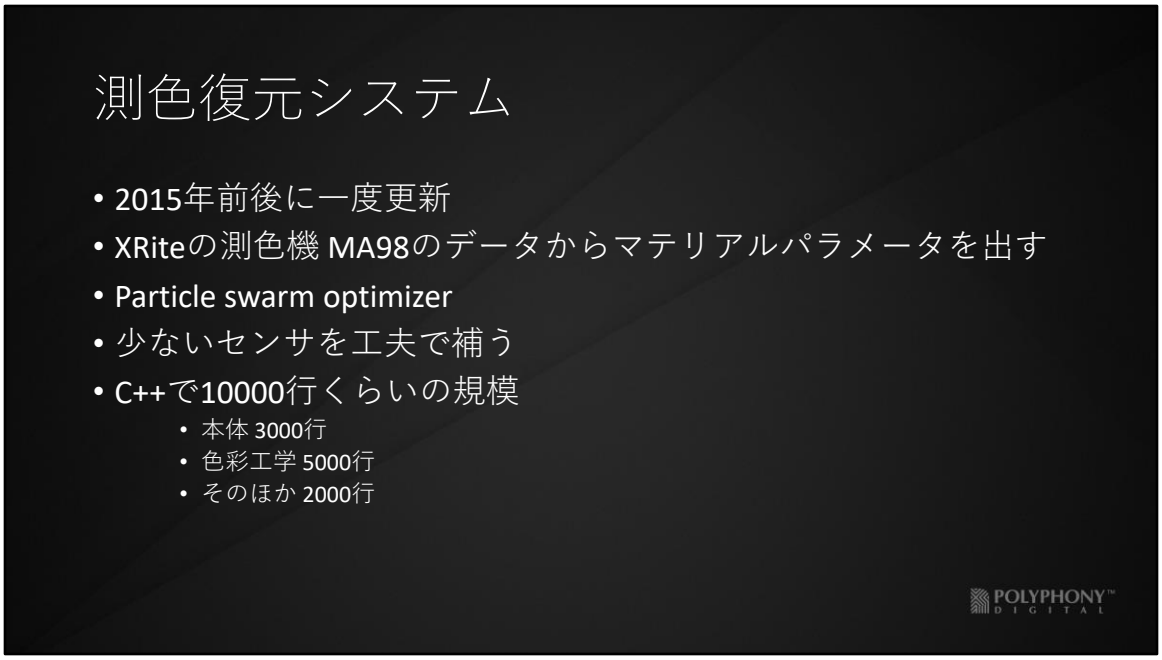
[Fukuda2010]

POLYPHONY DIGITAL

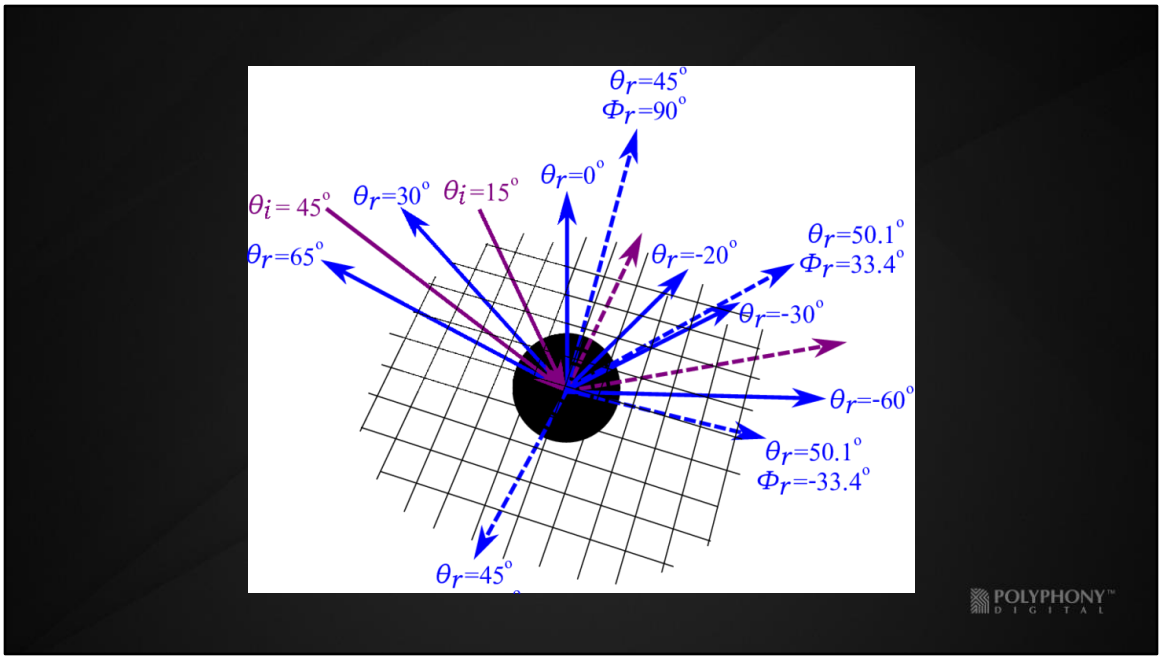
この、計測したBRDFデータから、マテリアルパラメータを推定するような問題は、
マテリアル推定、パラメータ推定などと呼ばれ多くの研究が行われてきました。
弊社でも、微分可能レンダリングなどが一般的になる以前から、
マテリアルパラメータの推定システムを開発、利用してきました。
社内ではこれを、測色復元、と呼んでいます。



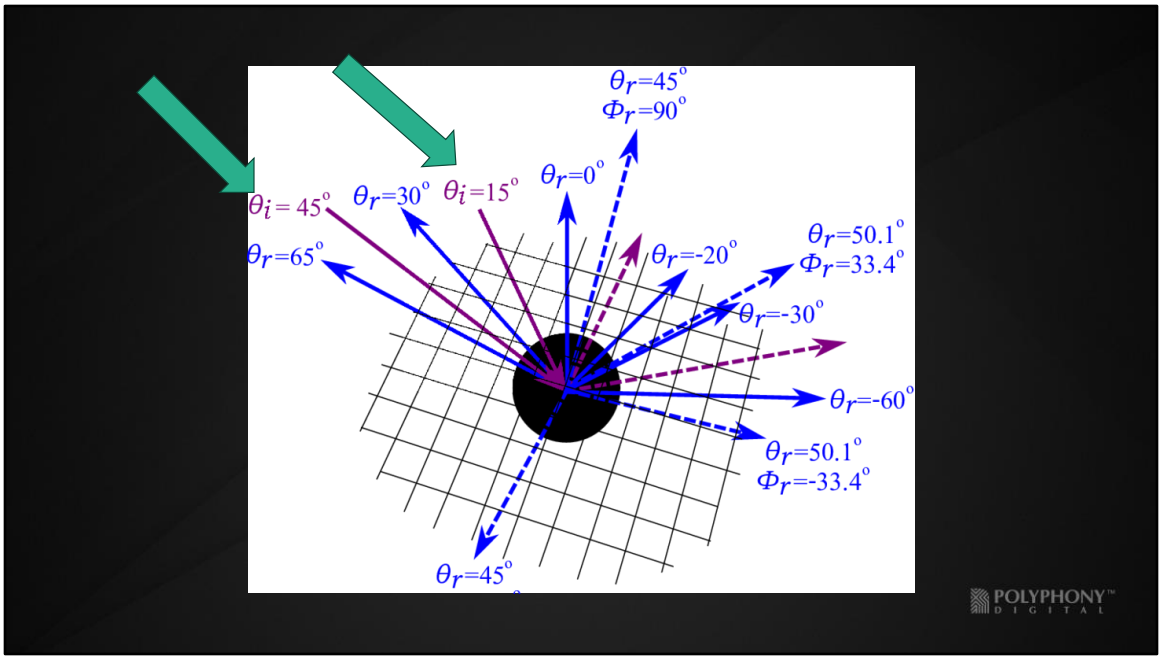
ここからは、弊社が元々開発利用していた、測色復元システムについて紹介します。



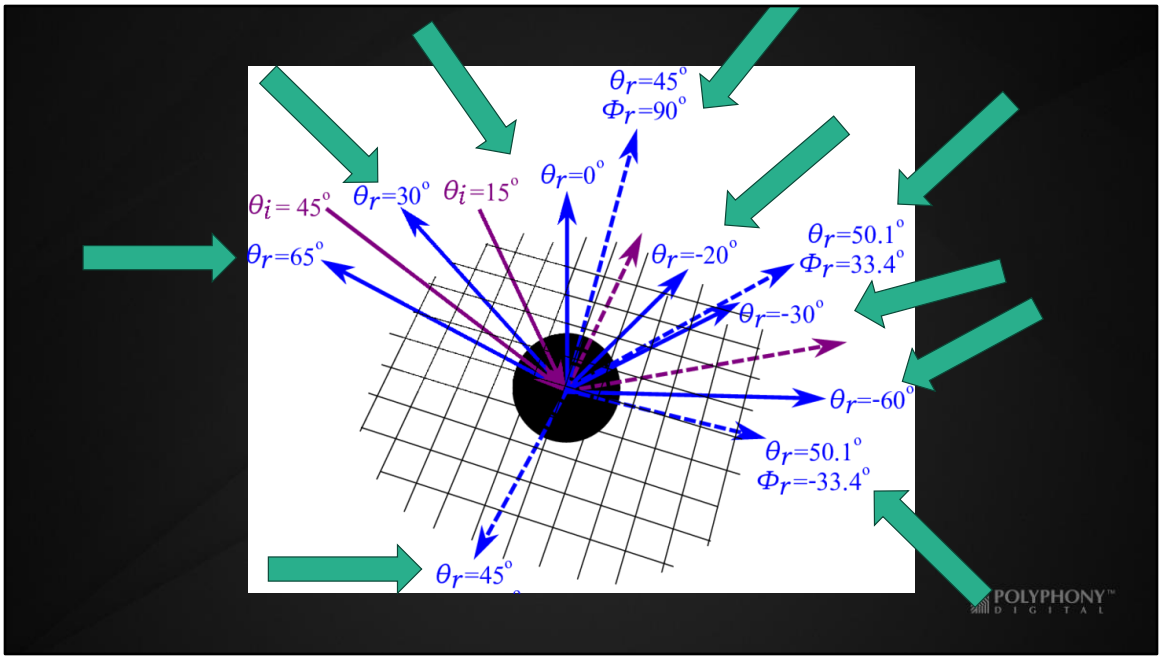
かなり以前から測色機での計測による測色復元システムを開発運用してきました。
XRiteのMA98という測色機を用いて計測したデータから、シェーダのパラメータを逆算するというものです。
2015年に一度書き直して、リニューアルしました。
実装はほぼフルスクラッチのC++で、CPU動作です。
最適化手法にはParticle Swarm Optimizerを用いていました。
MA98の計測データはサンプルが非常に疎だったため工夫で補う必要がありました。



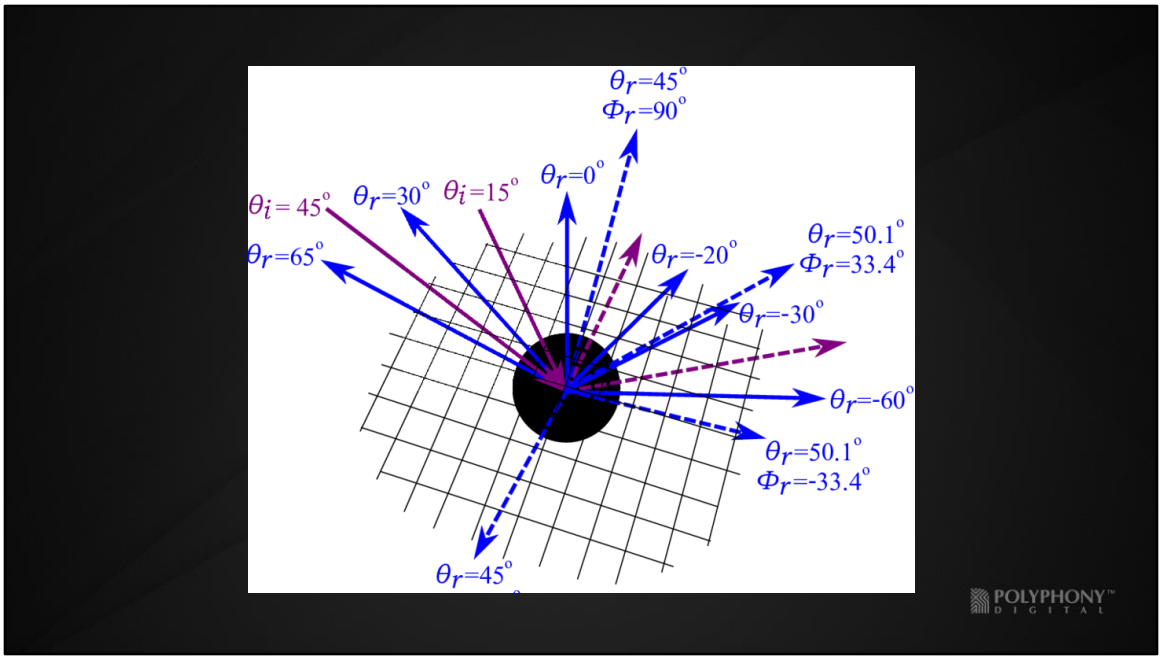
当時利用していた測色機　MA98は、この図のような角度の組み合わせで、サンプルを計測することができました。



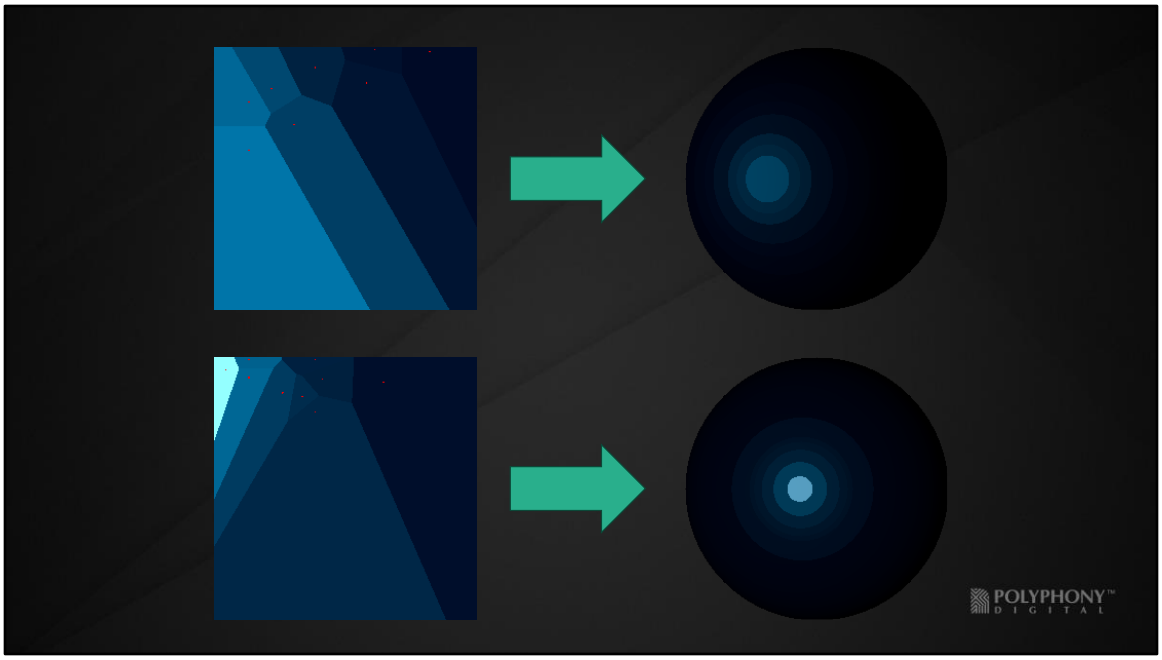
光の入射方向として、15度と45度の2方向



光を受光する分光センサがこの10方向にあります。



この2*10=20という、わずかなサンプルの組み合わせを用いて直接マテリアル推定をするプログラムを書いてみたところ、全く安定しないことがわかりました。



なんとかして解が求められるような手法を検討したところ、
仮想の半球上に計測データをマッピングし、
色差を求める手法に辿り着きました。

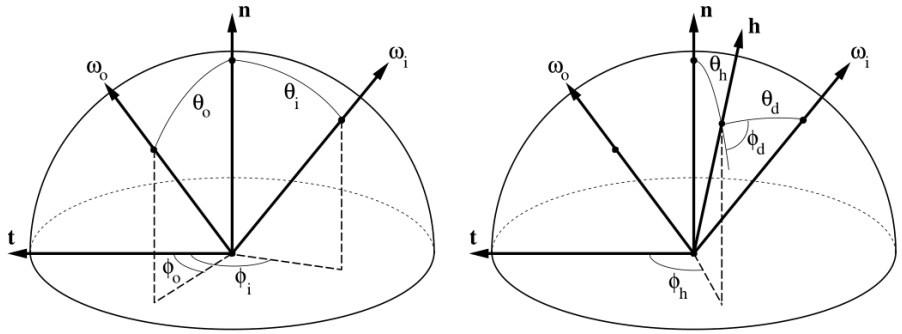
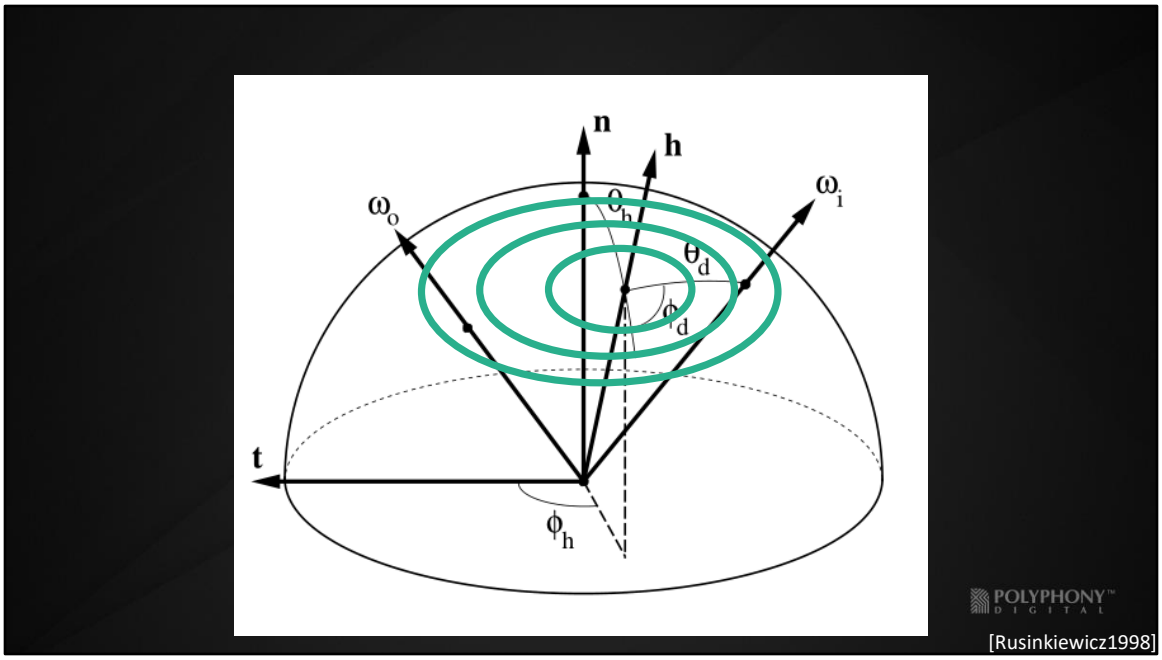


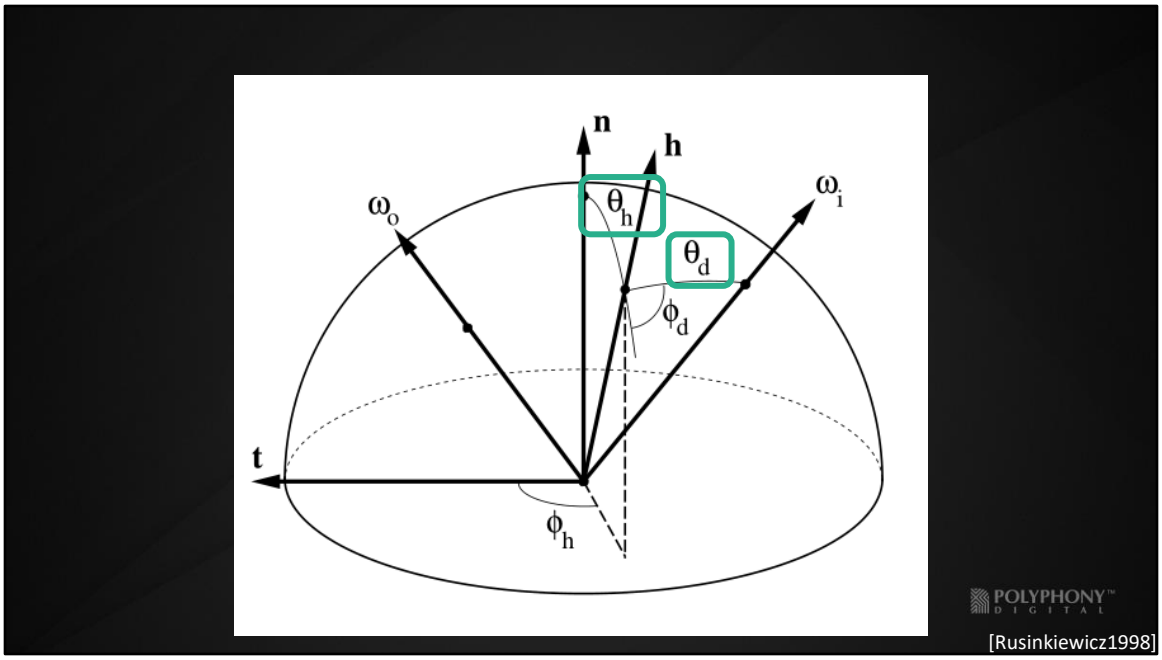
Figure 2: Proposed reparameterization of BRDFs. Instead of treating the BRDF as a function of (θ_i, ϕ_i) and (θ_o, ϕ_o), as shown on the left, we consider it to be a function of the halfangle (θ_h, ϕ_h) and a difference angle (θ_d, ϕ_d), as shown on the right. The vectors marked $\vec{n}$ and $\vec{t}$ are the surface normal and tangent, respectively.

[Rusinkiewicz1998]

わずかなサンプリング結果から、どうやってそれなりの仮想レンダリング結果を求めるか。
試行錯誤したところ、Rusinkieiwicz座標系で最近傍探索を行うのが良さそうだ、とわかりました。



Rusinkiewicz座標系はBRDFの入射方向と出射方向の組み合わせを、ハーフベクトルへの極座標を用いたかたちに変数変換する手法です。



特に、$\theta h, \theta d$で2次元平面に展開したBRDFは、BRDFスライス、と呼ばれ、あるBRDFの特徴を把握しやすいプロット方式として知られています。

BRDF slice

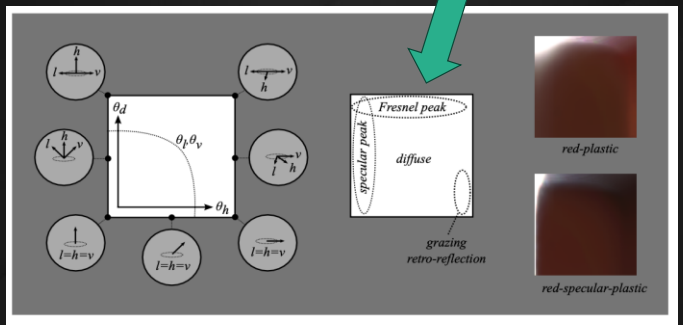
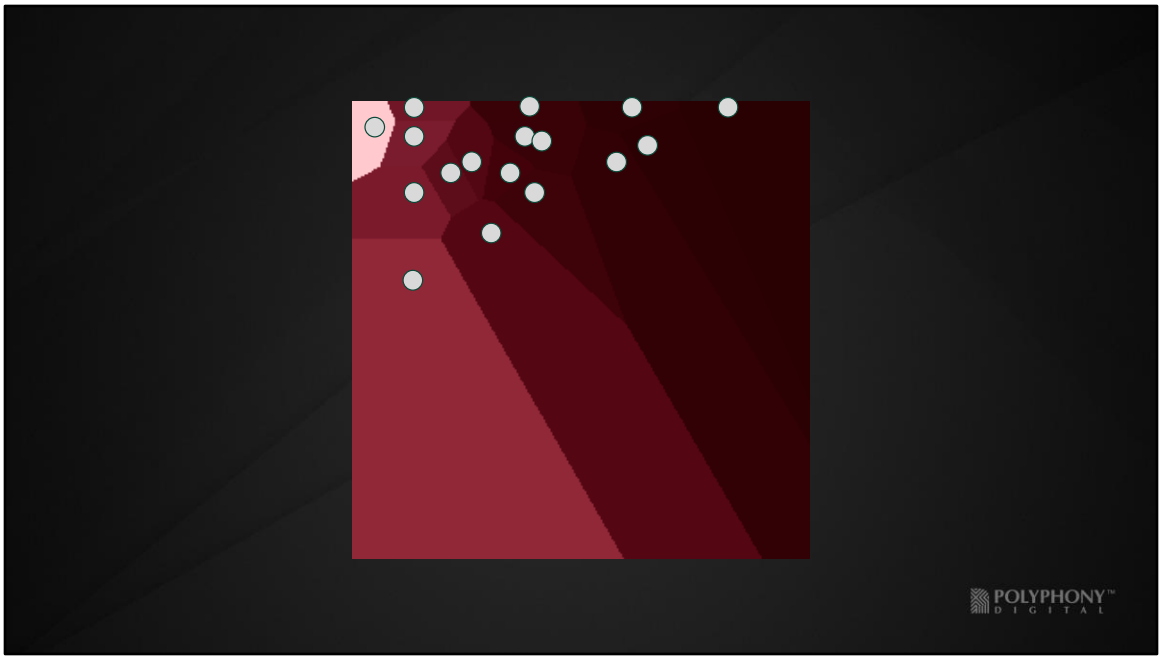


Figure 3: BRDF images slices for *red-plastic* and *specular-red-plastic* shown along with schematic view of "slice space."

[Burley2012]

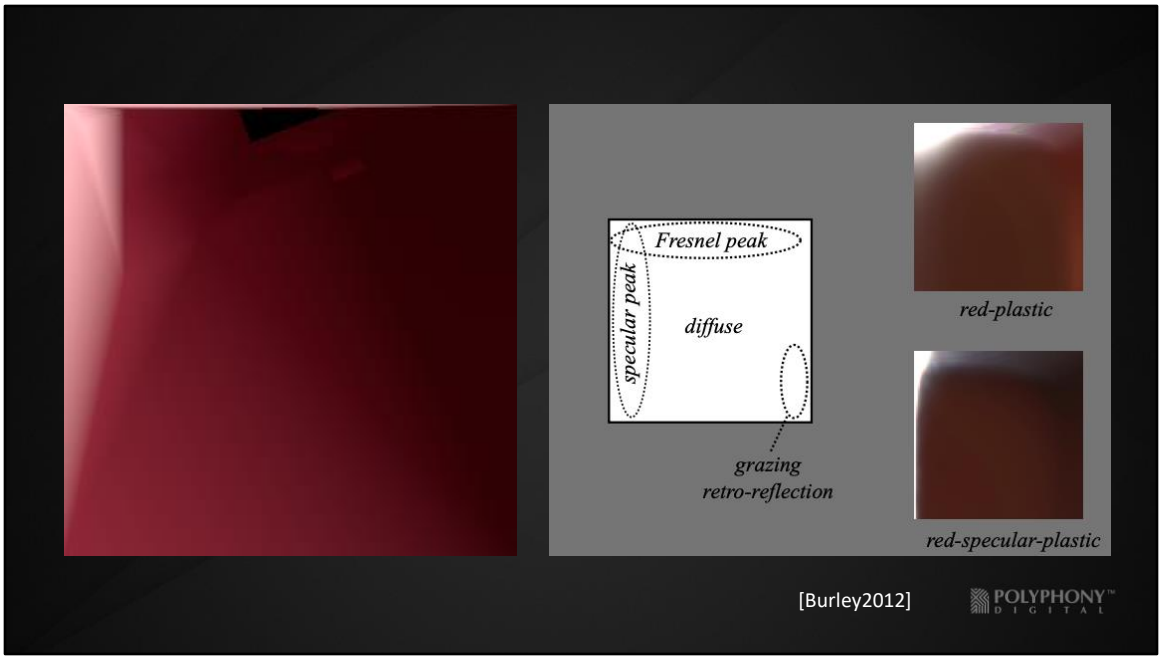
POLYPHONY DIGITAL

BRDFスライスには、BRDFの特徴が出やすいエリアが何箇所か存在します。フレネル反射の強さや、スペキュラの強さ、再帰反射の強さなどが一覧できます。

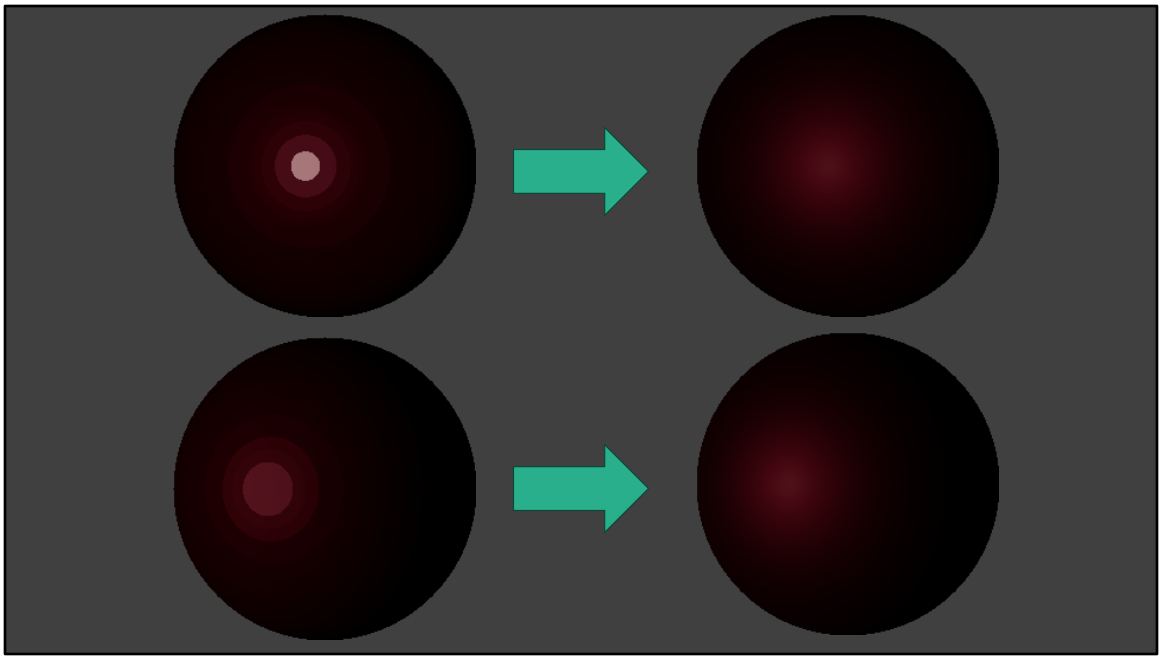


実際にMA98で測色した赤い車の塗料サンプルからBRDFスライスをプロットするとこのようになります。
画像中にある点が、サンプリングできているデータの位置です。
計測できているエリアがかなり限られていることがわかるでしょうか。

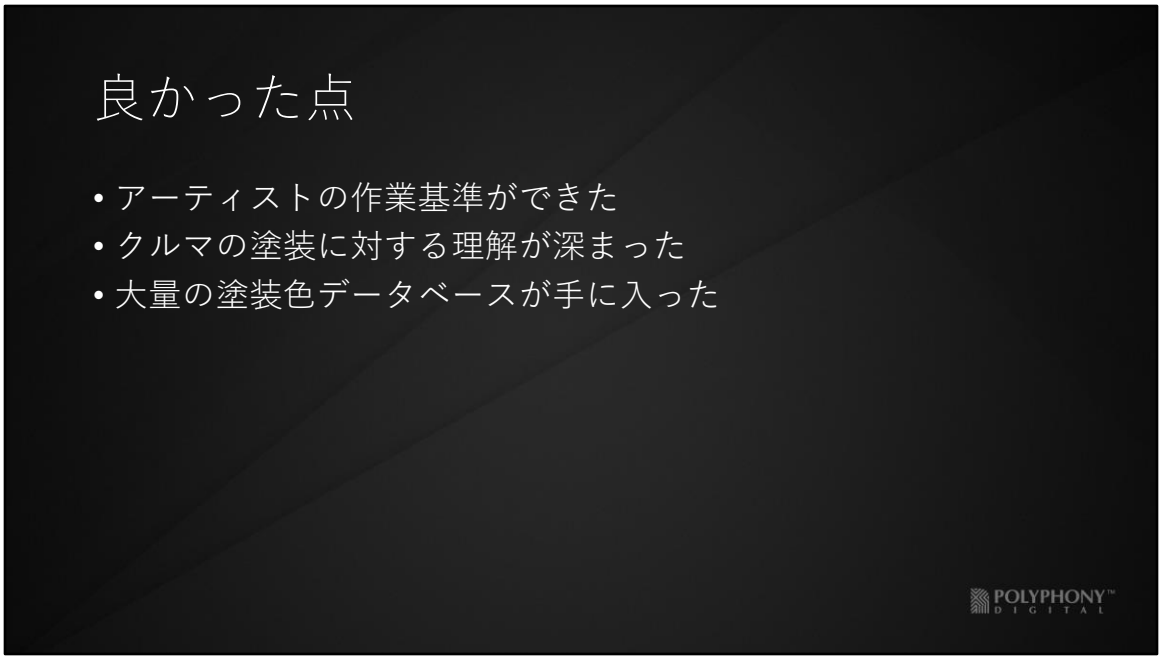
最近傍点ではなく、ドロネー分割して補間するとこのようなBRDFスライスになりますが、



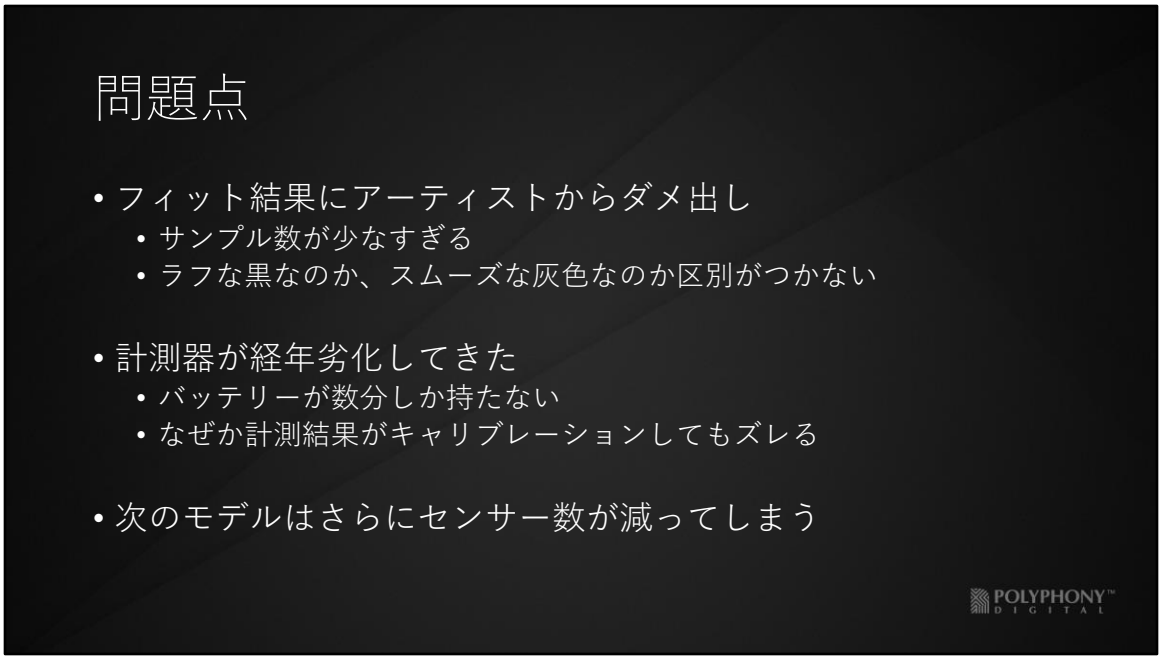
正しく計測されているBRDFのスライスと比較すると、どうしても不足しているのは否めません。



このBRDFスライス座標系で、サンプル点をルックアップすることによって、なんとか半球画像を捻り出し、色差をもとめ、なんとか測色復元ができるようになりました。
当時の知識と経験ではこの精度が精一杯でした。

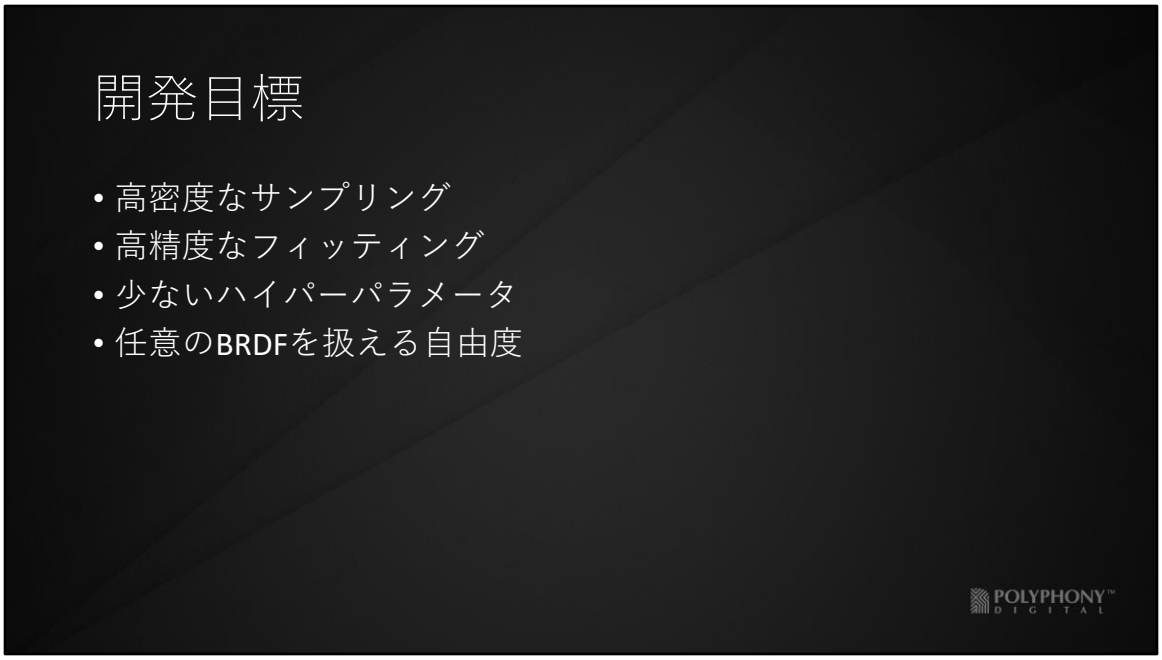


この測色復元システムによるメリットは以下の通りです。
何よりもまず、計測データに基づいて作業を行うことで、アーティストの作業基準ができました。
また、クルマの塗装についての理解を深めることにもなりました。
取材のたびに計測し収集した分光データベースは、かなりの規模にもなりました。



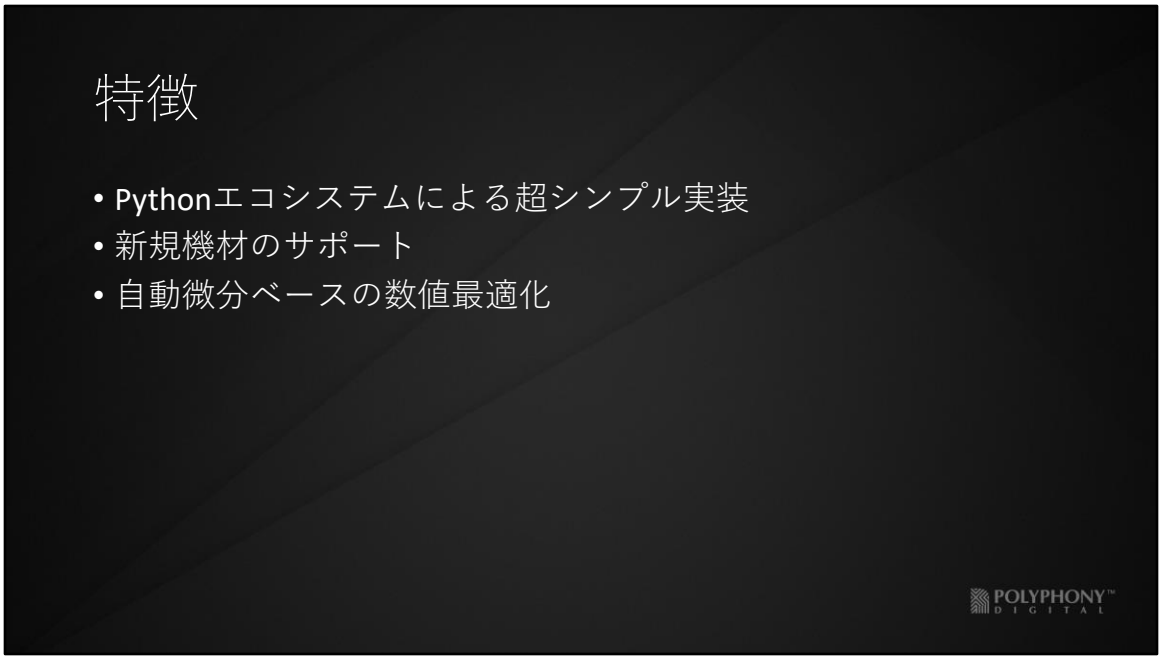
しかし、日が経つにつれ、問題点も現れてきました。

フィット結果にアーティストからのダメ出しが出るたびにハイパーパラメタを微調整するような日々が続いてしまいました。
特に、サンプル数が少な過ぎることによって、マットブラックとつややかな灰色の区別がつきにくい問題がなかなか解決できませんでした。
というのも、パラメータ探索に用いていたParticle Swarm Optimizationが局所解に陥りやすかったのです。
さらに、機材が物理的に経年劣化してしまいました。
バッテリーがほとんど持たなくなり、キャリブレーションや定期メンテナンスを行なってもなぜか計測結果がズレるような現象が発生しました。
機材更新を考えましたが、センサー数がさらに減ってしまうことがわかりました。



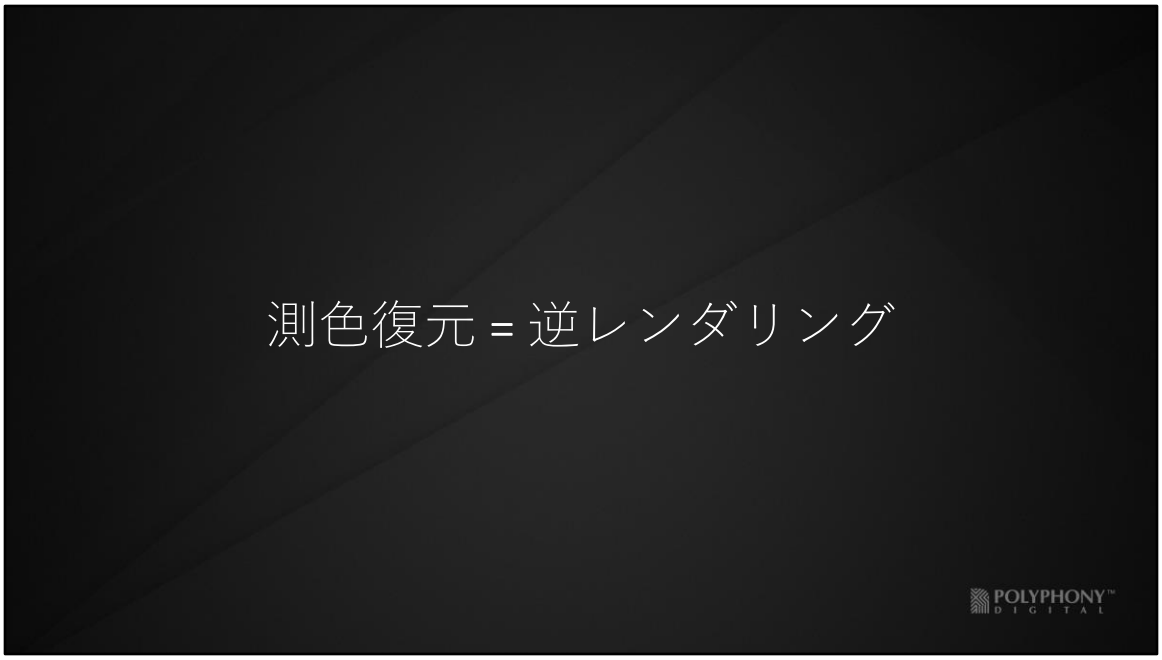
そこで、利用する機材を選定しなおし、
測色復元システムを新規に開発し直す決断をしました。
新規システムの開発目標は主に以下の 4 点です。
高密度なサンプリング。
高精度なフィッテイング。
少ないハイパーパラメータ。
そして、任意のBRDFを扱える自由度です。

ここからは、
新規に開発した測色復元システムについて紹介します。



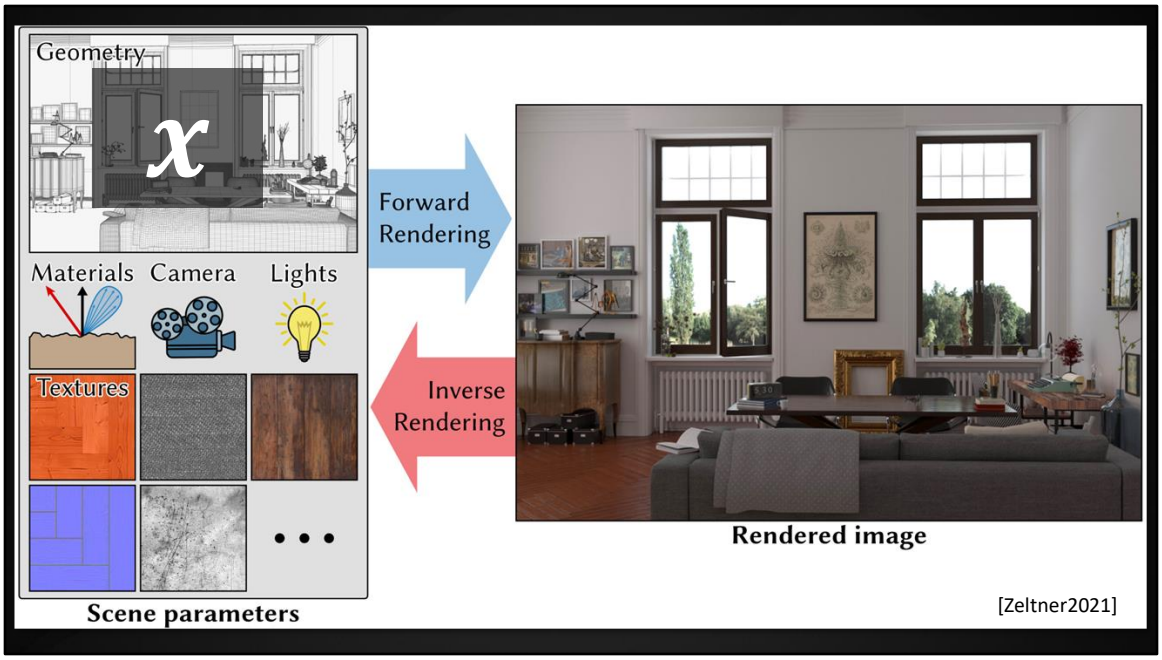
前回の測色復元システムが抱えてしまった技術的負債の原因をふまえ、
全面的にPythonでの実装とすることにしました。
これは、多くの機械学習や数値最適化のエコシステムが存在したため、その知見を最大限活用したかったためです。
また、利用する計測機器を更新し、より高精度、高密度なBRDF計測を可能にしました。
最も大きなポイントは、本セッションのタイトルにもありますが、
微分可能レンダリングや、そのベースとなっている自動微分を用いて数値最適化としてパラメータ推定を行なっている点です。

少し寄り道になりますが、
微分と、自動微分と、微分可能レンダリングについて紹介させてください。

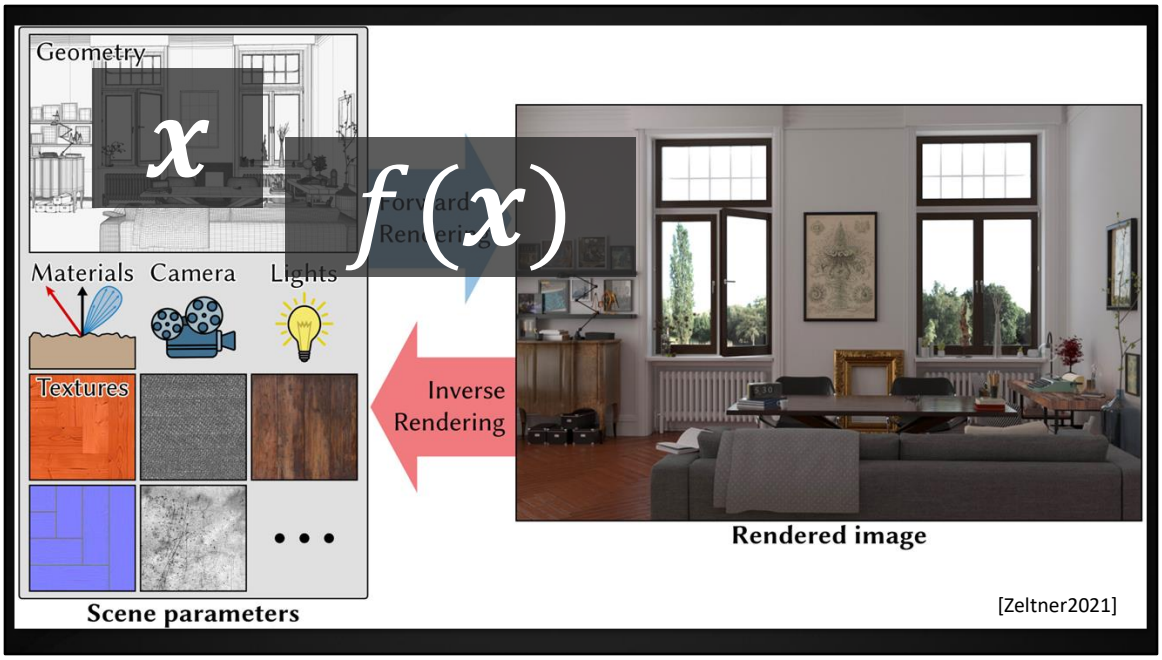


測色復元はすなわち、与えられた画像からシーン内のパラメータを求めるという逆レンダリング問題の一つです。

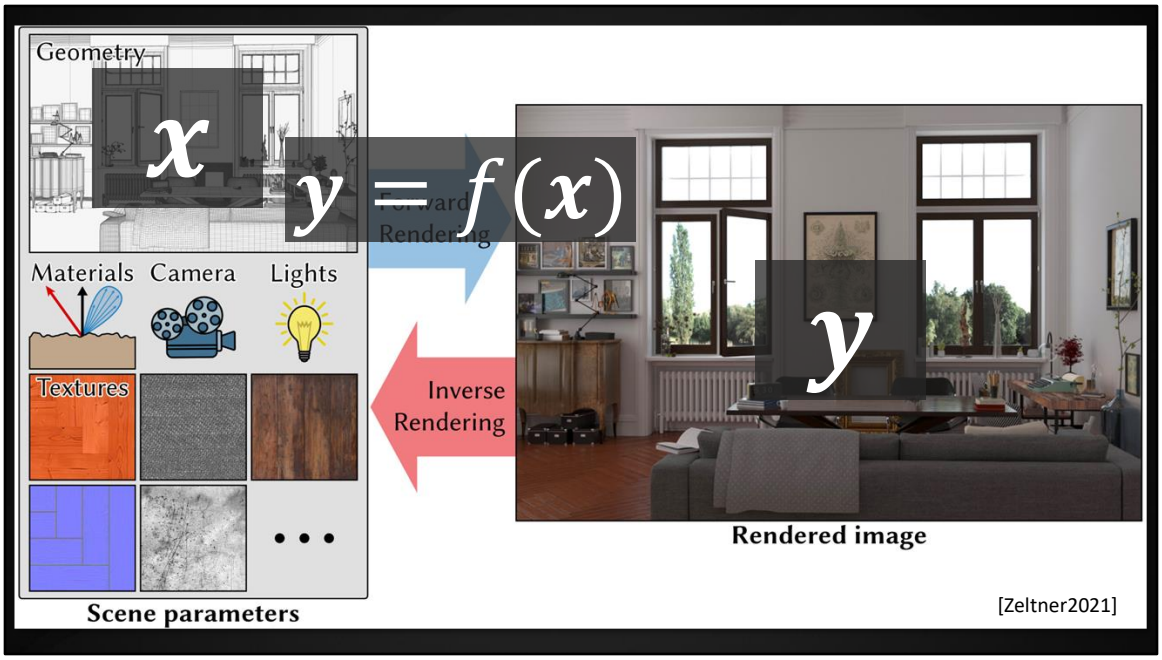
微分可能レンダリングは、逆レンダリング問題を解くための重要な一手法です。



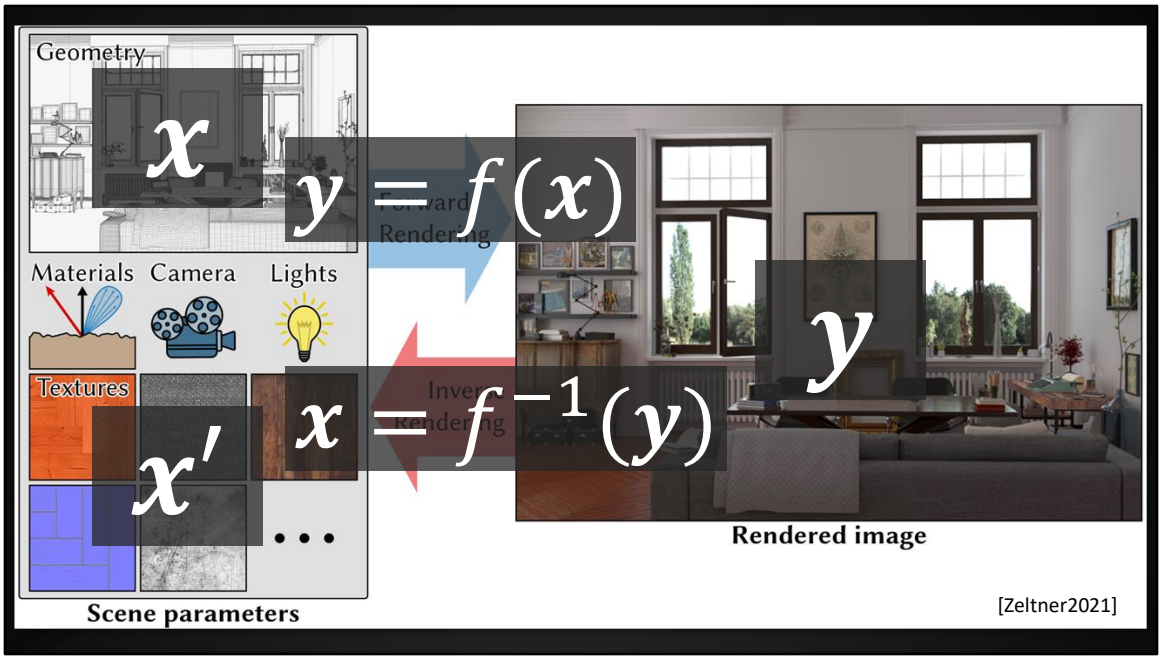
あるシーンとそのパラメータを、変数 x とします。



すると、レンダリングするという行為は、
レンダリング関数 f に、引数 x を与える、という行為に他なりません。



ここでは、レンダリングした結果の画像を、y としましょう。



$$\boldsymbol{x}$$

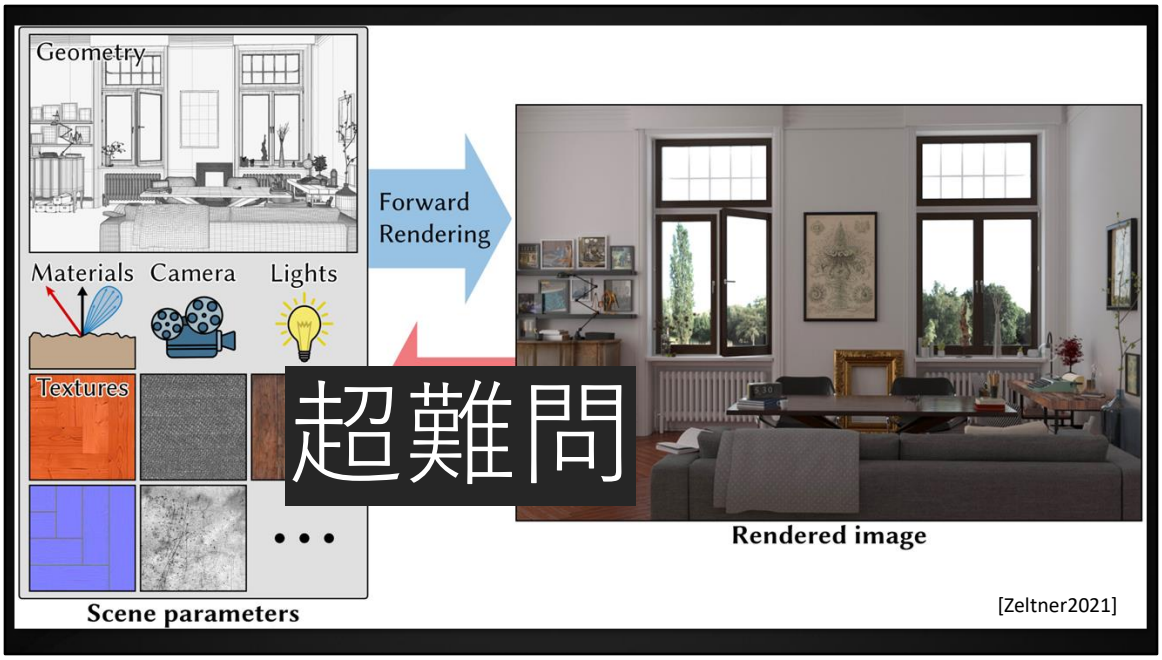
$$\boldsymbol{y} = f(\boldsymbol{x})$$

$$\boldsymbol{y}$$

$$\boldsymbol{x} = f^{-1}(\boldsymbol{y})$$

$$\boldsymbol{x}'$$

この逆レンダリング関数を簡単に求めることができれば、
あるシーンを再現できるようなパラメータを求める問題は、簡単に解くことができるはずです。



ところがこれが超難問なわけです。

逆関数なしに解を求めたい

$$y = f(x)$$

$$x = ?$$

POLYPHONY DIGITAL

逆関数を求めずに、答えがyになるようなxを求めたい。
どうすれば良いでしょうか？

誤差を考える

$$y = f(x)$$
$$y' = f(x')$$

$$g = y' - y$$

まず適当にもしかしたら解かもしれない変数 x' を用意し、
関数を計算してみましょう。
すると、正解の値との間に、誤差 g を計算することができます。

誤差が0ってことは…

$$g = f(x') - f(x)$$

もし$g = 0$なら
$x' = x$と期待できる

もし、誤差が 0 になるのなら、その引数　x’ は少なくとも、解x のうちのひとつであると期待できます。

逆関数を求めずに誤差を最小化する

$$g = f(x') - f(x)$$

$$g \to 0$$
$$x' \to x$$

POLYPHONY DIGITAL

逆関数が求められないときに、何かの解を得たいなら、こうして、誤差を最小化するという問題に置き換えることができるわけです。
では、どうやって誤差を最小化するか。ここで利用するのが微分です。

微分とは

ある関数の、引数を変化した時の、値の変化量

$$y = f(x)$$

$$\dot{f}(x) = \frac{dy}{dx} = \frac{d}{dx} f(x)$$

POLYPHONY DIGITAL

微分は、ある関数の入力が変化した時に出力がどれくらい変化するか、という値です。

導関数

$$y = f(x)$$

$$\dot{f}(x) = \frac{dy}{dx} = \frac{d}{dx} f(x)$$

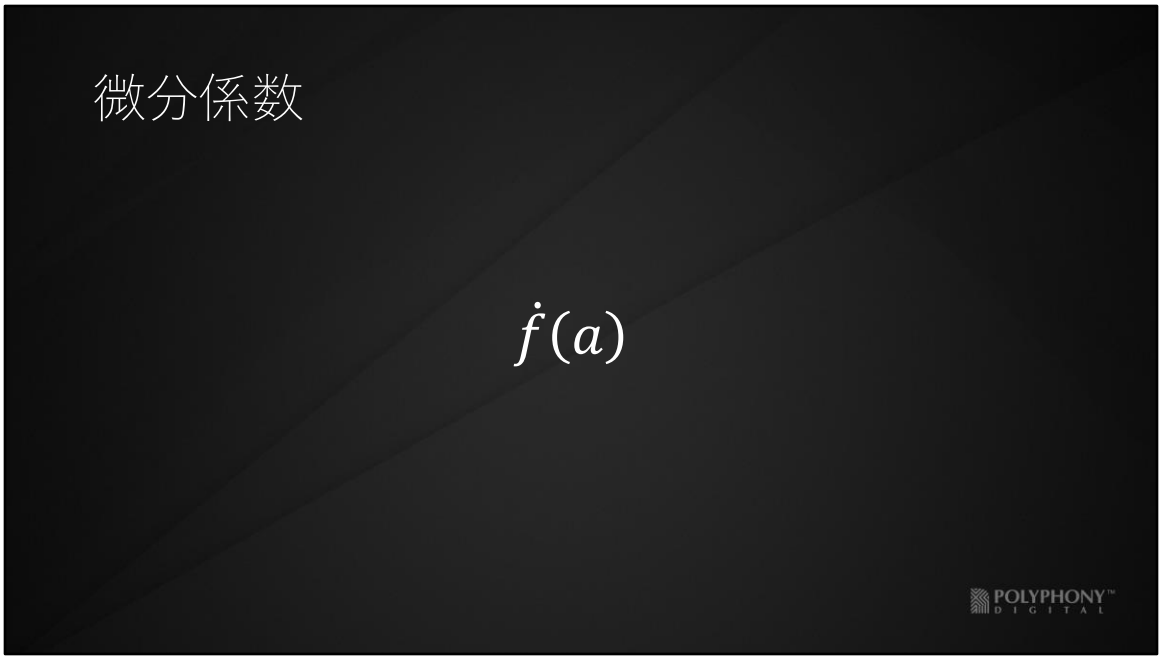
この、ある関数を微分した関数を、導関数と呼びます。
数式ではこうして上にドットをつけたりして表すことがあります。

微分する＝導関数を求める

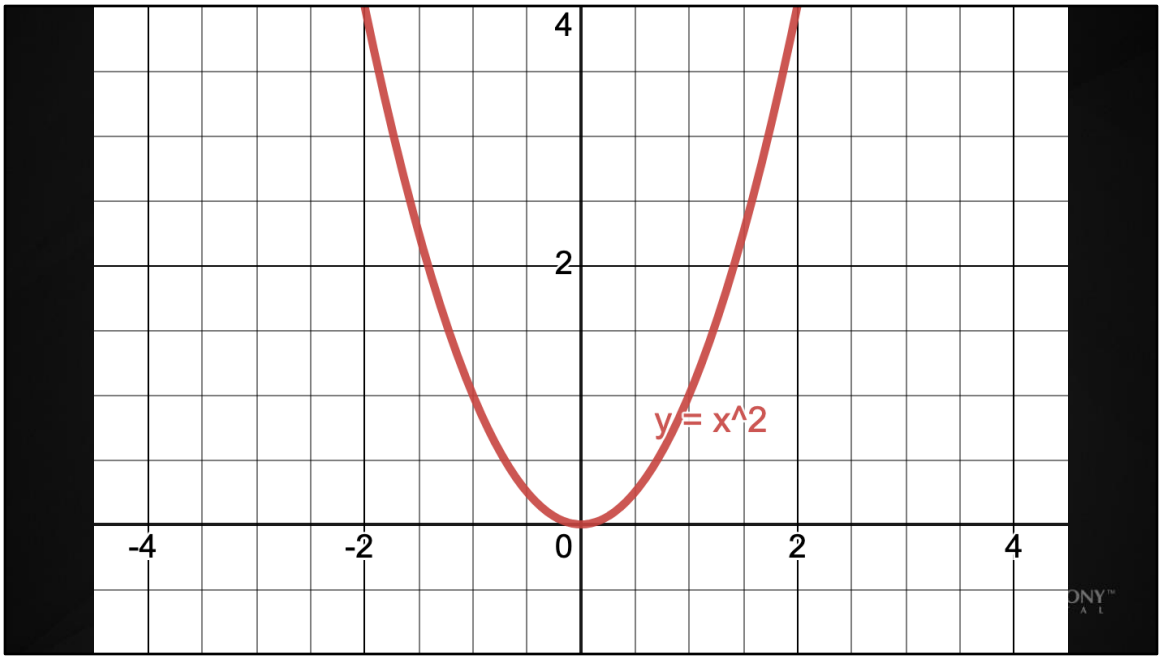
$$\dot{f}(x) = \frac{dy}{dx} = \frac{d}{dx} f(x)$$

POLYPHONY DIGITAL

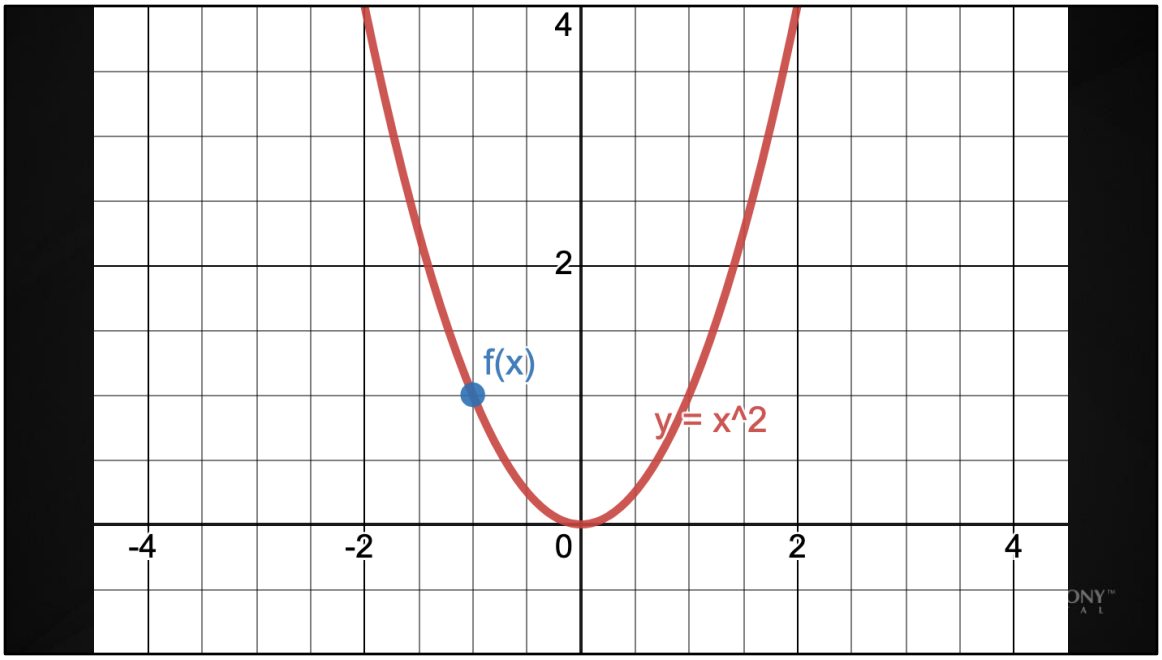
ある関数に対して、その導関数を求めることを、微分する、と言います。



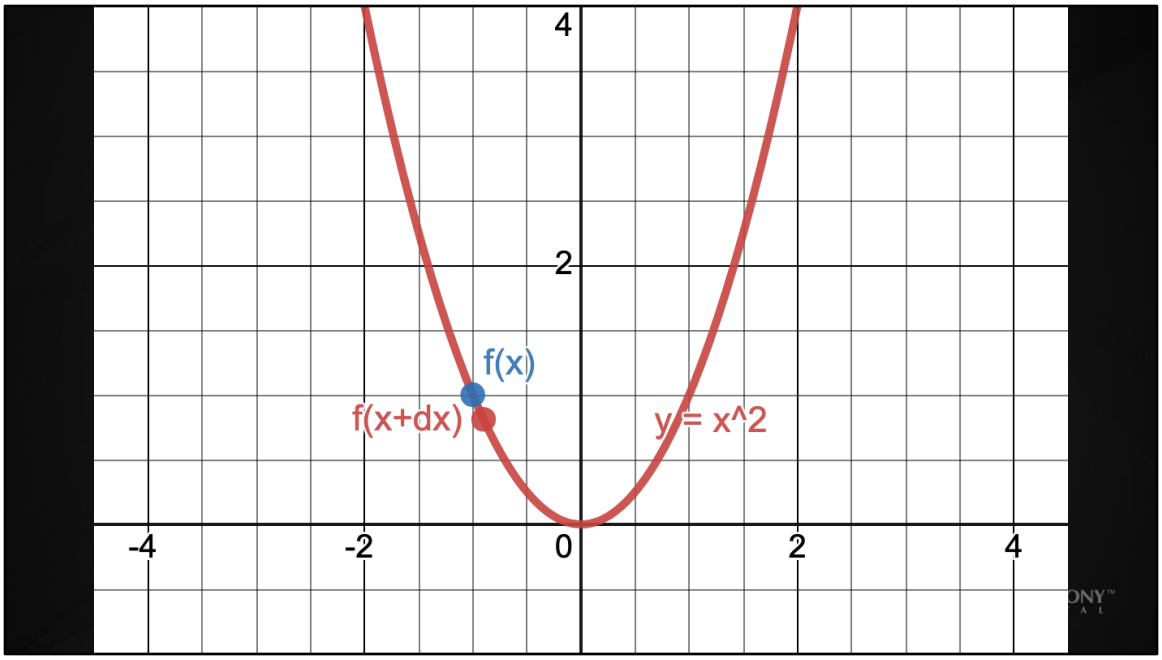
また、求めた導関数の値を　任意の点で求めた時、その値を、微分係数　と呼びます。



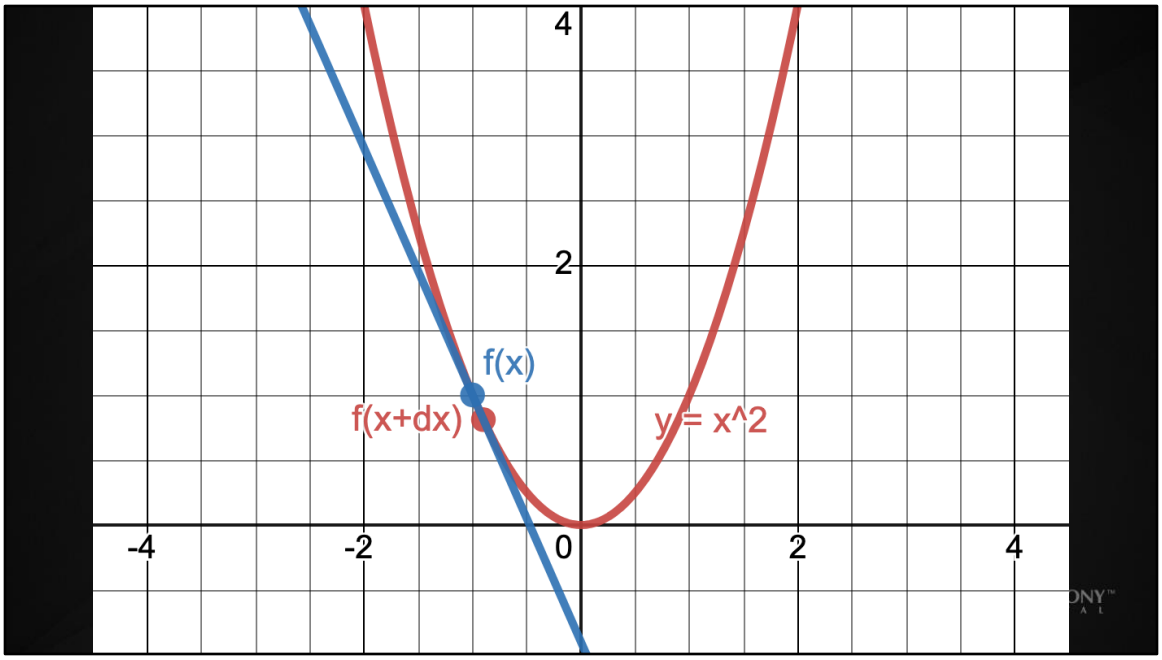
シンプルな関数の微分係数を求めてみましょう。



微分したい点の値と、



わずかにずらした点の二点を考えて、



この二つの点を通る線の、傾きを考えます。これが、この関数の、この点における、微分係数です。

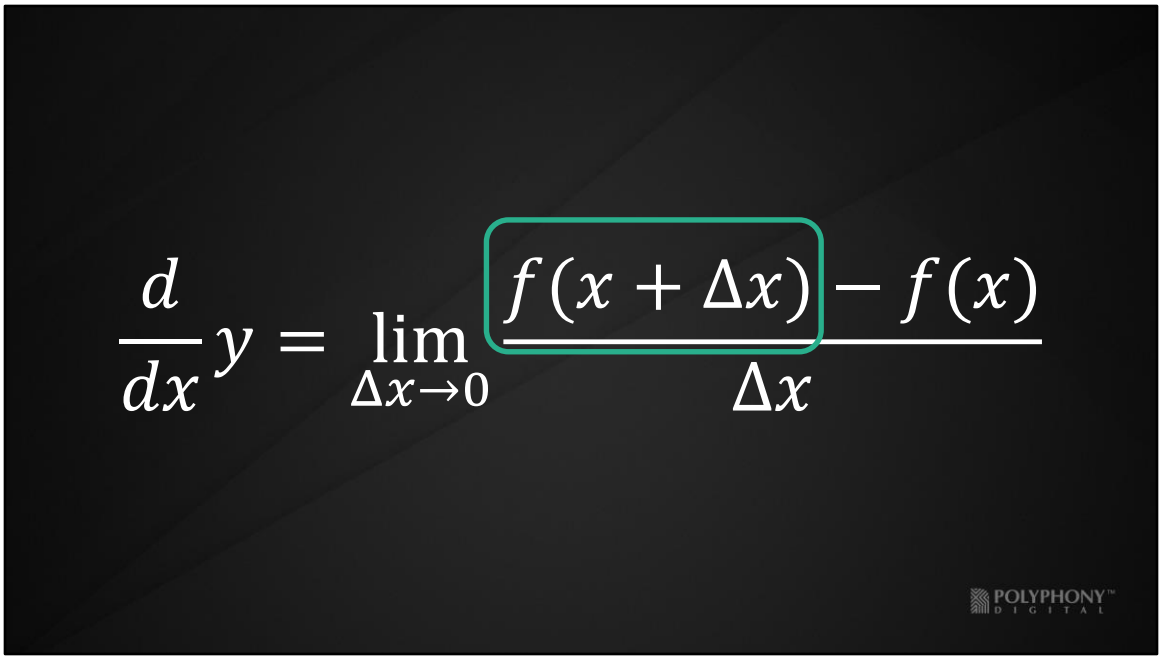
$$y = f(x)$$

の微分は

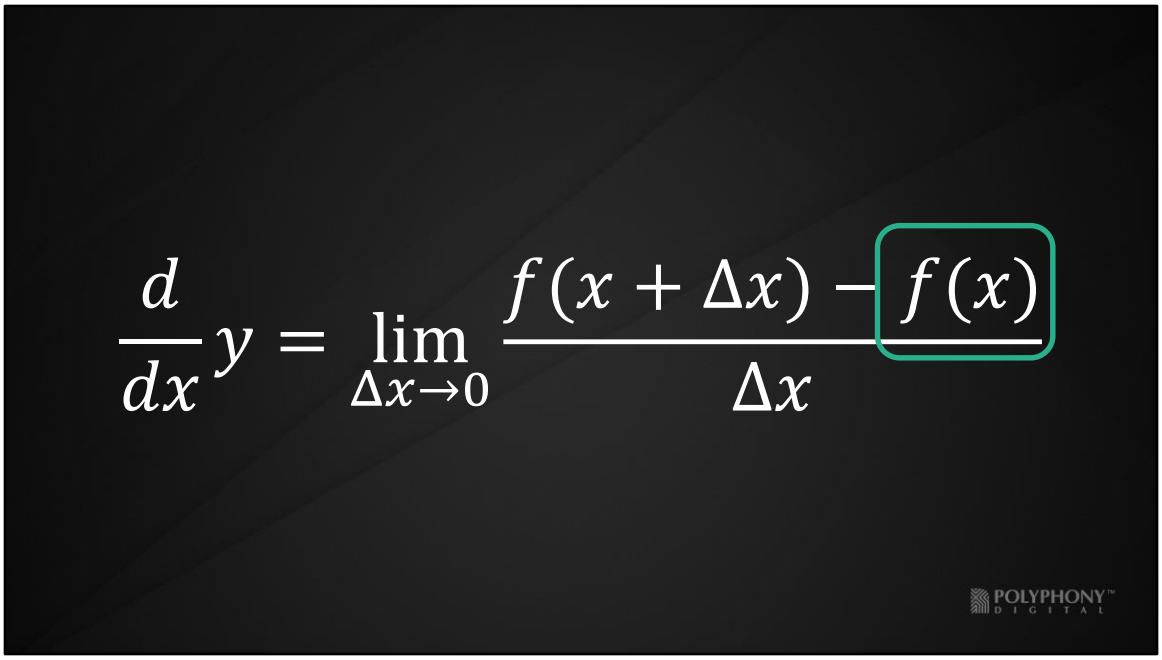
$$\frac{d}{dx}y = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

POLYPHONY™ DIGITAL

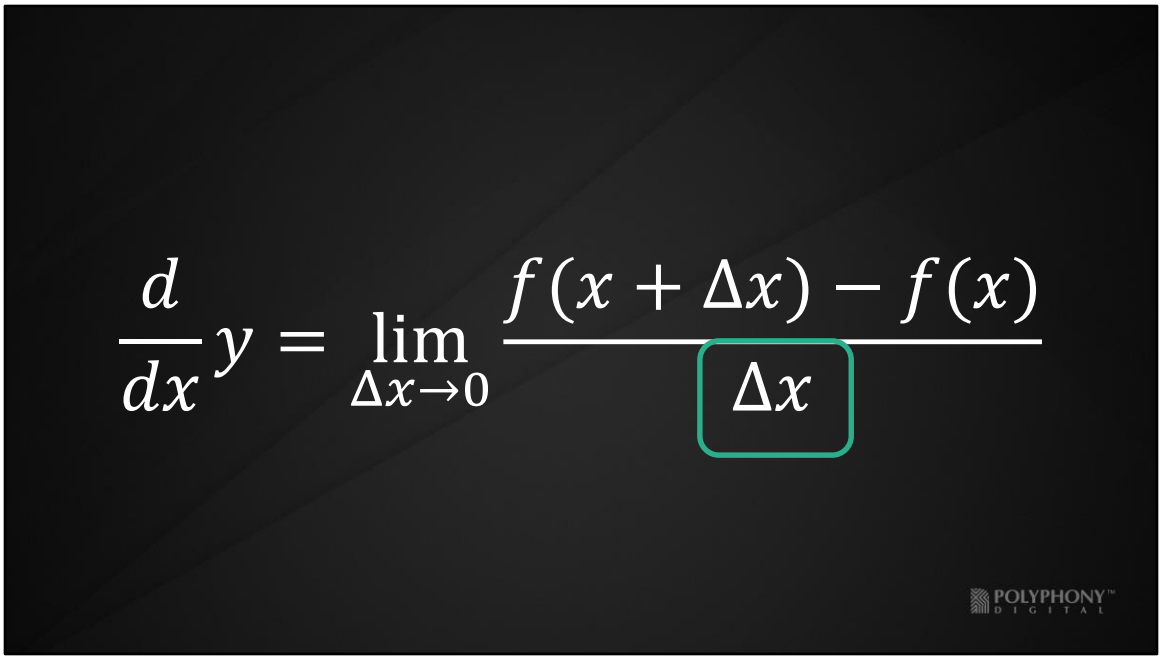
この、小さな値をつかって傾きを求めることを、数学的に書くとこのようになります。



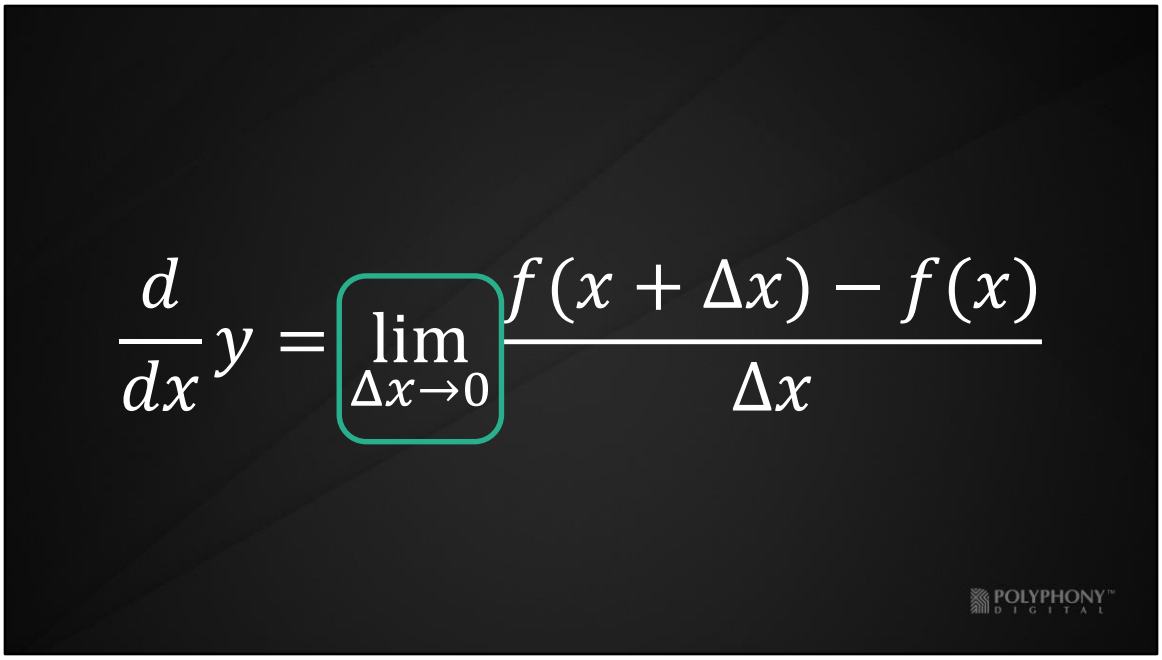
小さな値Δx方向にちょっとずらした値から


$\frac{d}{dx}y = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$
POLYPHONY DIGITAL

もとの値を引いて


$\frac{d}{dx}y = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$
POLYPHONY DIGITAL

傾きを求めます。



この式に対して、小さな値Δxを 0 に近づけていった極限が導関数です。

誤差が0なら、それは解のうちの一つ

$$g = f(x') - f(x)$$

もし$g = 0$なら
$x' = x$と期待できる

POLYPHONY DIGITAL

先ほど、逆関数を求めなくても、誤差が 0 なら、それは解のうちのひとつである、という説明をしました。

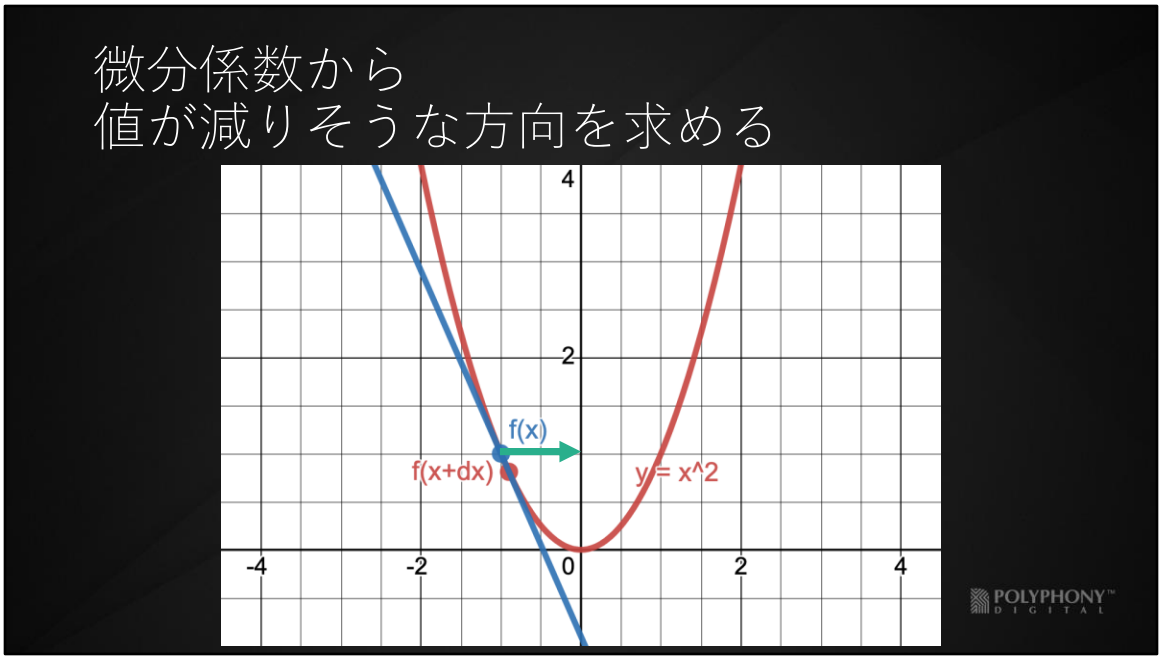
解を求めるために、関数の最小値を探す

$$g = f(x') - f(x)$$

$$g \to 0$$
$$x' \to x$$

POLYPHONY DIGITAL

今考えている問題は逆問題が難しいという前提があり、
いきなり誤差が 0 の点を探すことはできません。
そこで、誤差を最小化して、解の候補 x'を 真の解 x に近づけていくような手法を考えます。



先ほどのこの図を見るように、微分係数を計算すれば、値が減りそうな方向を見つけることができます。

勾配法

$$x_{t+1} = x_t - \eta \frac{d}{dx} f(x)$$

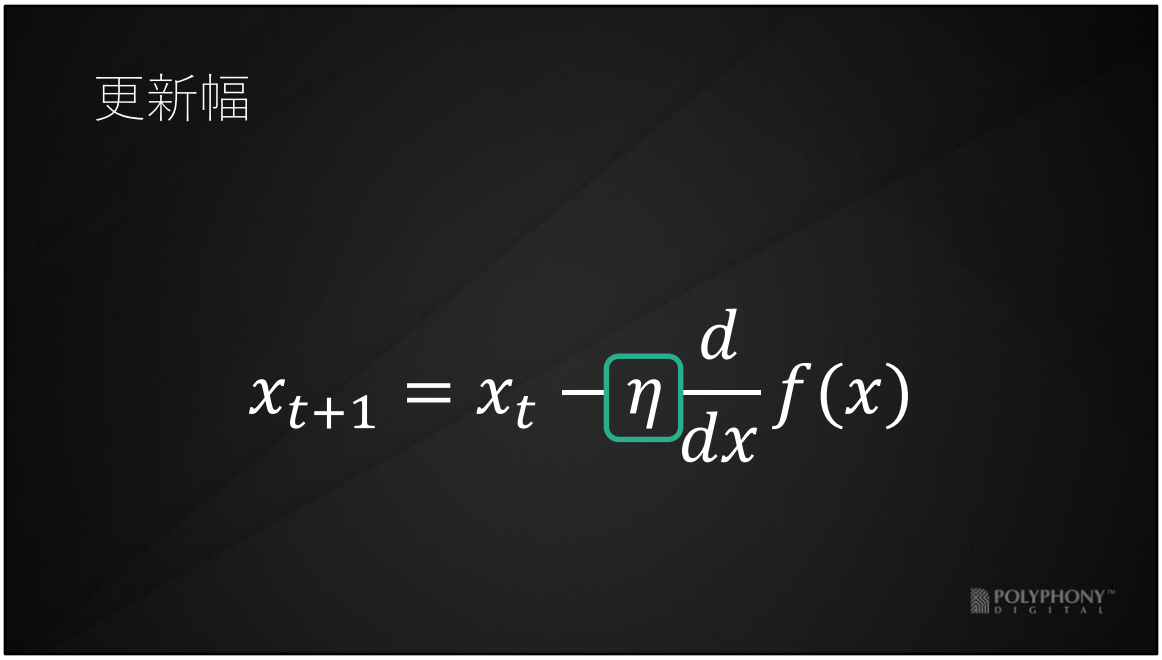
この、値が変化しそうな方向を、勾配と呼びます。
勾配を用いて解を求める手法を、勾配法と言います。
数式にするとこのようになります。

繰り返し手法

$$x_{t+1} = x_t - \eta \frac{d}{dx} f(x)$$

POLYPHONY DIGITAL

値が減りそうな方向に繰り返し進んで、解を求めるという、繰り返し型の手法です。



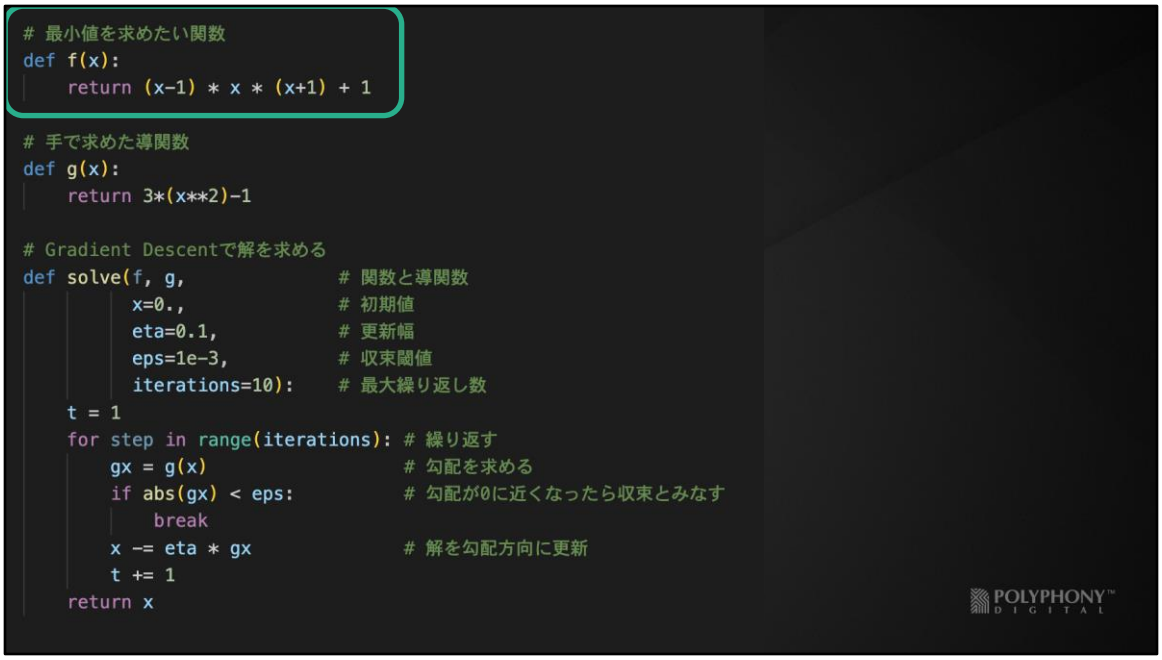
イータは更新幅で、大きくすると高速に解がもとまる一方、計算が安定しにくくなるというトレードオフにあります。

最急降下法

```python
# 最小値を求めたい関数
def f(x):
    return (x-1) * x * (x+1) + 1

# 手で求めた導関数
def g(x):
    return 3*(x**2)-1

# Gradient Descentで解を求める
def solve(f, g,              # 関数と導関数
          x=0.,              # 初期値
          eta=0.1,           # 更新幅
          eps=1e-3,          # 収束閾値
          iterations=10):    # 最大繰り返し数
    t = 1
    for step in range(iterations): # 繰り返す
        gx = g(x)                  # 勾配を求める
        if abs(gx) < eps:          # 勾配が0に近くなったら収束とみなす
            break
        x -= eta * gx              # 解を勾配方向に更新
        t += 1
    return x
```

POLYPHONY DIGITAL

これをプログラムで表すと、このようになります。
この手法を、勾配法の中でも、最急降下法と呼びます。

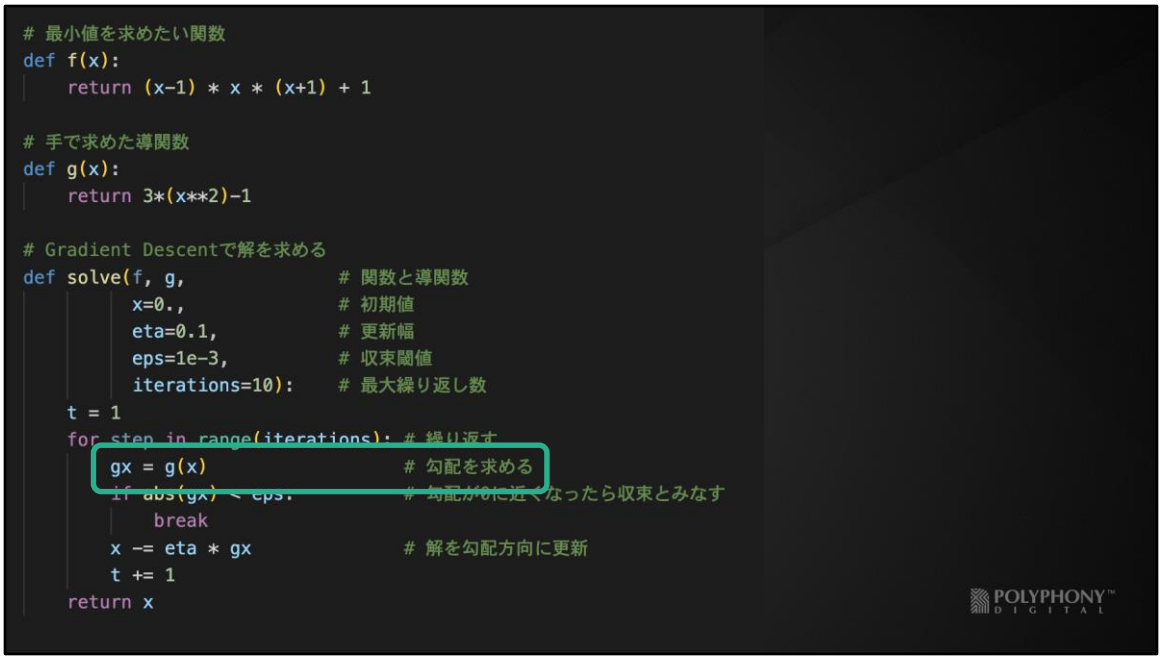



```python
# 最小値を求めたい関数
def f(x):
    return (x-1) * x * (x+1) + 1

# 手で求めた導関数
def g(x):
    return 3*(x**2)-1

# Gradient Descentで解を求める
def solve(f, g,              # 関数と導関数
          x=0.,              # 初期値
          eta=0.1,           # 更新幅
          eps=1e-3,          # 収束閾値
          iterations=10):    # 最大繰り返し数
    t = 1
    for step in range(iterations): # 繰り返す
        gx = g(x)                  # 勾配を求める
        if abs(gx) < eps:          # 勾配が0に近くなったら収束とみなす
            break
        x -= eta * gx              # 解を勾配方向に更新
        t += 1
    return x
```

まず、最小値を求めたい関数を定義します。

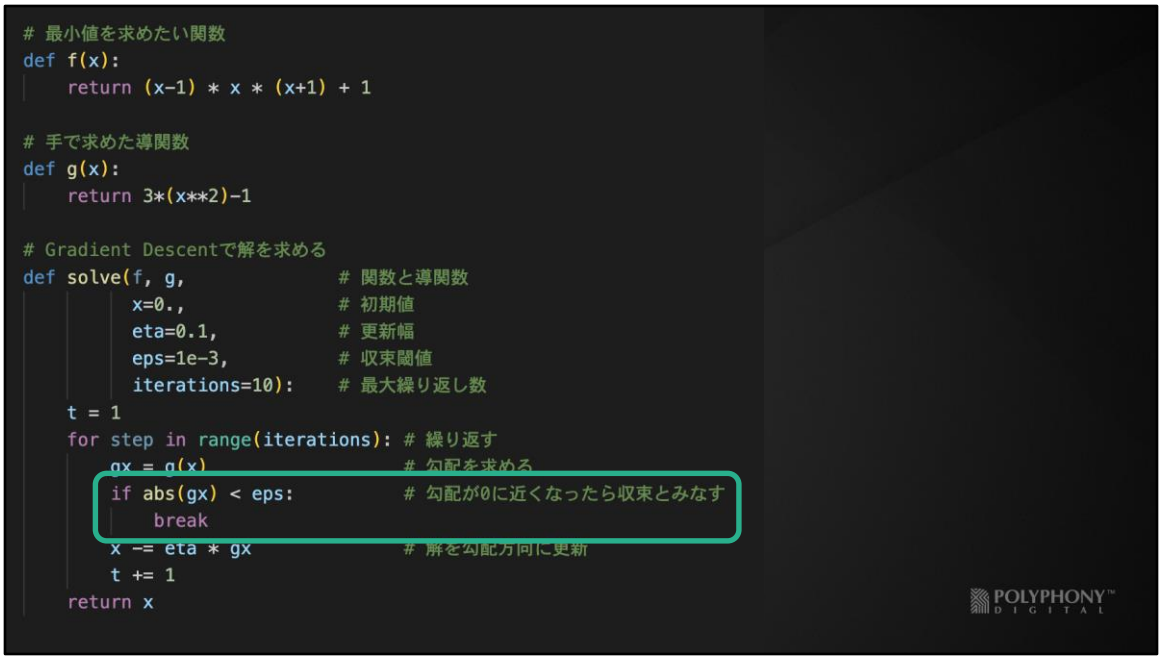
```python
# 最小値を求めたい関数
def f(x):
    return (x-1) * x * (x+1) + 1

# 手で求めた導関数
def g(x):
    return 3*(x**2)-1

# Gradient Descentで解を求める
def solve(f, g,              # 関数と導関数
          x=0.,              # 初期値
          eta=0.1,           # 更新幅
          eps=1e-3,          # 収束閾値
          iterations=10):    # 最大繰り返し数
    t = 1
    for step in range(iterations): # 繰り返す
        gx = g(x)                  # 勾配を求める
        if abs(gx) < eps:          # 勾配が0に近くなったら収束とみなす
            break
        x -= eta * gx              # 解を勾配方向に更新
        t += 1
    return x
```


その導関数は、今回は手で求めました。



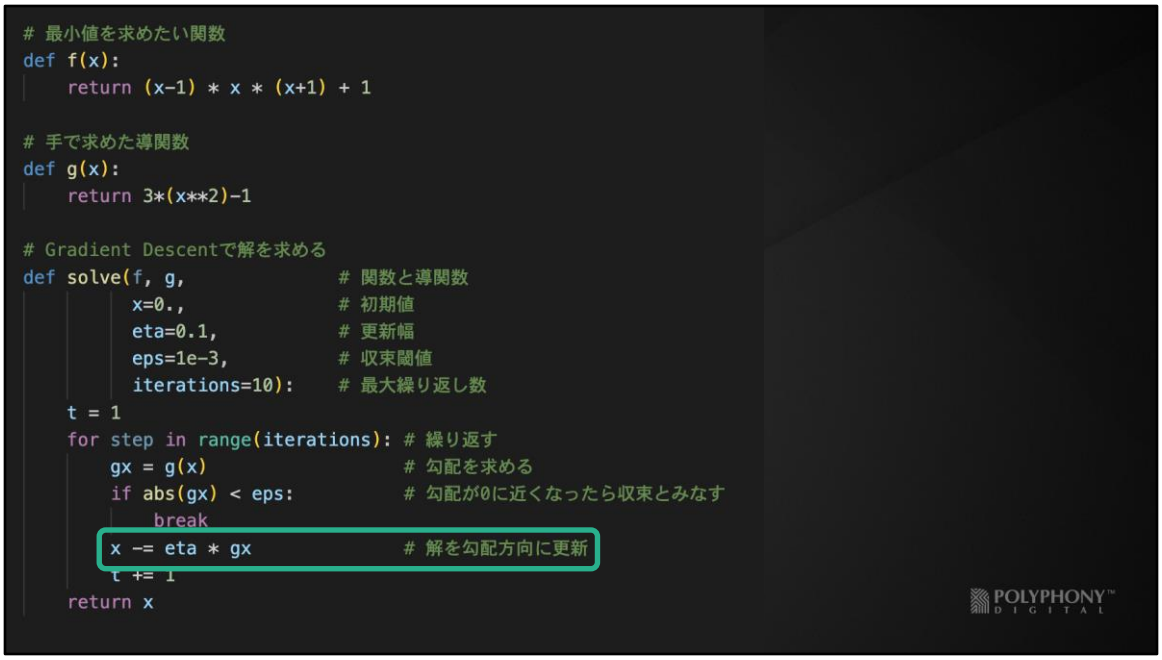

```python
# 最小値を求めたい関数
def f(x):
    return (x-1) * x * (x+1) + 1

# 手で求めた導関数
def g(x):
    return 3*(x**2)-1

# Gradient Descentで解を求める
def solve(f, g,              # 関数と導関数
          x=0.,              # 初期値
          eta=0.1,           # 更新幅
          eps=1e-3,          # 収束閾値
          iterations=10):    # 最大繰り返し数
    t = 1
    for step in range(iterations): # 繰り返す
        gx = g(x)                  # 勾配を求める
        if abs(gx) < eps:          # 勾配が0に近くなったら収束とみなす
            break
        x -= eta * gx              # 解を勾配方向に更新
        t += 1
    return x
```

解を求めるループの中では、まず微分係数から勾配を求めます。



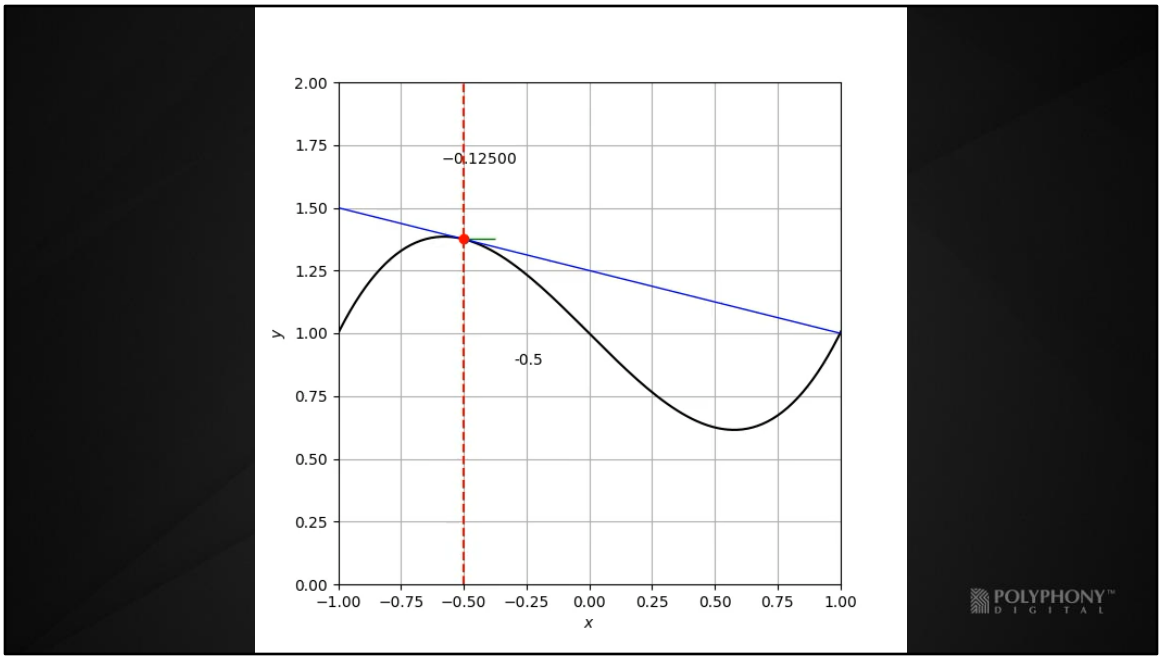

```python
# 最小値を求めたい関数
def f(x):
    return (x-1) * x * (x+1) + 1

# 手で求めた導関数
def g(x):
    return 3*(x**2)-1

# Gradient Descentで解を求める
def solve(f, g,              # 関数と導関数
          x=0.,              # 初期値
          eta=0.1,           # 更新幅
          eps=1e-3,          # 収束閾値
          iterations=10):    # 最大繰り返し数
    t = 1
    for step in range(iterations): # 繰り返す
        gx = g(x)                  # 勾配を求める
        if abs(gx) < eps:          # 勾配が0に近くなったら収束とみなす
            break
        x -= eta * gx              # 解を勾配方向に更新
        t += 1
    return x
```

勾配が 0 に近くなった場合、収束したとみなしてループを抜けます。

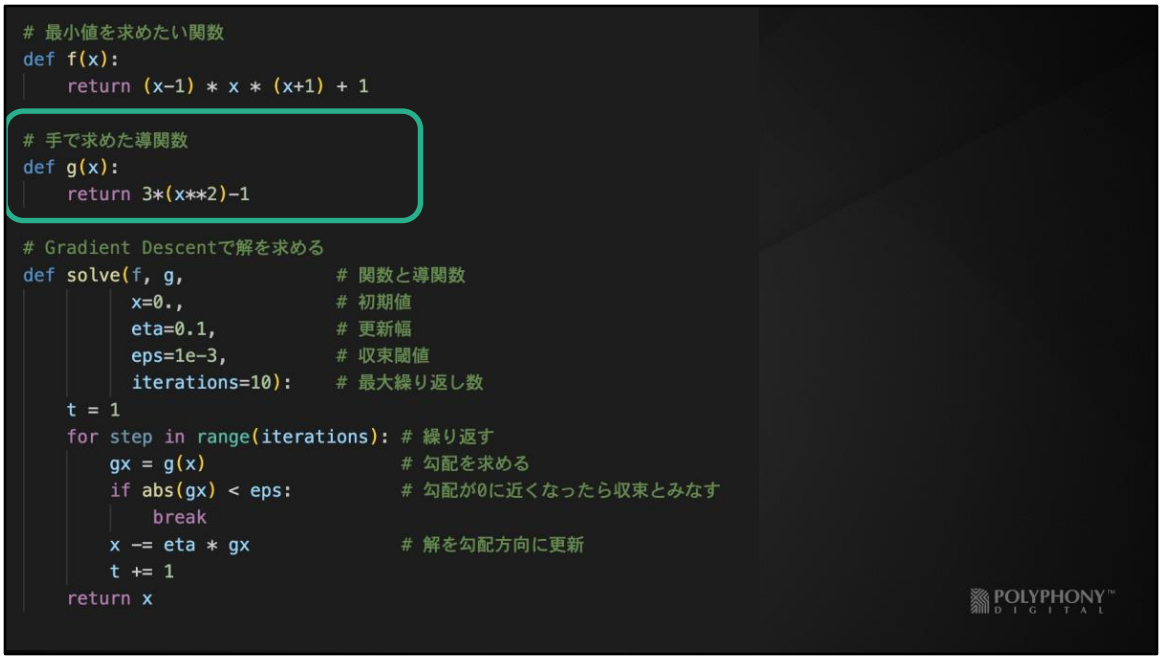



```python
# 最小値を求めたい関数
def f(x):
    return (x-1) * x * (x+1) + 1

# 手で求めた導関数
def g(x):
    return 3*(x**2)-1

# Gradient Descentで解を求める
def solve(f, g,              # 関数と導関数
          x=0.,              # 初期値
          eta=0.1,           # 更新幅
          eps=1e-3,          # 収束閾値
          iterations=10):    # 最大繰り返し数
    t = 1
    for step in range(iterations): # 繰り返す
        gx = g(x)                  # 勾配を求める
        if abs(gx) < eps:          # 勾配が0に近くなったら収束とみなす
            break
        x -= eta * gx              # 解を勾配方向に更新
        t += 1
    return x
```

まだ収束していない場合は、求めた勾配で解を更新していきます。



実行結果はこのようになります。
微分で求めた勾配によって、最小値を求めることができているのがわかります。




```python
# 最小値を求めたい関数
def f(x):
    return (x-1) * x * (x+1) + 1

# 手で求めた導関数
def g(x):
    return 3*(x**2)-1

# Gradient Descentで解を求める
def solve(f, g,              # 関数と導関数
          x=0.,              # 初期値
          eta=0.1,           # 更新幅
          eps=1e-3,          # 収束閾値
          iterations=10):    # 最大繰り返し数
    t = 1
    for step in range(iterations): # 繰り返す
        gx = g(x)                  # 勾配を求める
        if abs(gx) < eps:          # 勾配が0に近くなったら収束とみなす
            break
        x -= eta * gx              # 解を勾配方向に更新
        t += 1
    return x
```

今回は手で導関数を求めましたが、
もっと複雑な数式の場合、毎回これをやるのは大変です。

自動微分フレームワーク

- 計算式に対して、その微分を自動的に求める仕組み
  - PyTorch, autodiff, etc..
- 計算グラフを記録しておいて、バックトラックする
- Pythonは別に必須ではない
  - が、エコシステムは大きい

[autodiff]
[pytorch]
[github:micrograd]

POLYPHONY DIGITAL

一般のさまざまな数式やプログラムに対していちいち微分を求めることは非常に煩雑ですし、やりたくありません。
そこで使うのが自動微分です。
自動微分は、計算式に対して、自動的に微分係数を計算してくれる手法です。
近年の機械学習の進化に伴って、とてもたくさんのライブラリやフレームワークが出てきました。

```python
class Var:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []
        self.grad = 0

    def __add__(self, other):
        return Var(self.value + other.value, [(1, self), (1, other)])

    def __mul__(self, other):
        return Var(self.value * other.value, [(other.value, self), (self.value, other)])

    def sin(self):
        return Var(math.sin(self.value), [(math.cos(self.value), self)])

    def calc_grad(self, grad=1):
        self.grad += grad
        for coef, child in self.children:
            child.calc_grad(grad * coef)
```

[Wikipedia:AD]

Wikipediaに非常に良いサンプル実装があったので紹介します。

```python
class Var:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []
        self.grad = 0

    def __add__(self, other):
        return Var(self.value + other.value, [(1, self), (1, other)])

    def __mul__(self, other):
        return Var(self.value * other.value, [(other.value, self), (self.value, other)])

    def sin(self):
        return Var(math.sin(self.value), [(math.cos(self.value), self)])

    def calc_grad(self, grad=1):
        self.grad += grad
        for coef, child in self.children:
            child.calc_grad(grad * coef)
```

[Wikipedia:AD]

自動微分では、計算途中の変数を記録するため、変数クラスを用意します。

```python
class Var:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []
        self.grad = 0

    def __add__(self, other):
        return Var(self.value + other.value, [(1, self), (1, other)])

    def __mul__(self, other):
        return Var(self.value * other.value, [(other.value, self), (self.value, other)])

    def sin(self):
        return Var(math.sin(self.value), [(math.cos(self.value), self)])

    def calc_grad(self, grad=1):
        self.grad += grad
        for coef, child in self.children:
            child.calc_grad(grad * coef)
```

[Wikipedia:AD]

計算中の値そのものと

```python
class Var:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []
        self.grad = 0

    def __add__(self, other):
        return Var(self.value + other.value, [(1, self), (1, other)])

    def __mul__(self, other):
        return Var(self.value * other.value, [(other.value, self), (self.value, other)])

    def sin(self):
        return Var(math.sin(self.value), [(math.cos(self.value), self)])

    def calc_grad(self, grad=1):
        self.grad += grad
        for coef, child in self.children:
            child.calc_grad(grad * coef)
```

[Wikipedia:AD]

計算グラフの記録と

```python
class Var:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []
        self.grad = 0

    def __add__(self, other):
        return Var(self.value + other.value, [(1, self), (1, other)])

    def __mul__(self, other):
        return Var(self.value * other.value, [(other.value, self), (self.value, other)])

    def sin(self):
        return Var(math.sin(self.value), [(math.cos(self.value), self)])

    def calc_grad(self, grad=1):
        self.grad += grad
        for coef, child in self.children:
            child.calc_grad(grad * coef)
```

[Wikipedia:AD]

微分係数をメンバーに持ちます。

```python
class Var:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []
        self.grad = 0

    def __add__(self, other):
        return Var(self.value + other.value, [(1, self), (1, other)])

    def __mul__(self, other):
        return Var(self.value * other.value, [(other.value, self), (self.value, other)])

    def sin(self):
        return Var(math.sin(self.value), [(math.cos(self.value), self)])

    def calc_grad(self, grad=1):
        self.grad += grad
        for coef, child in self.children:
            child.calc_grad(grad * coef)
```

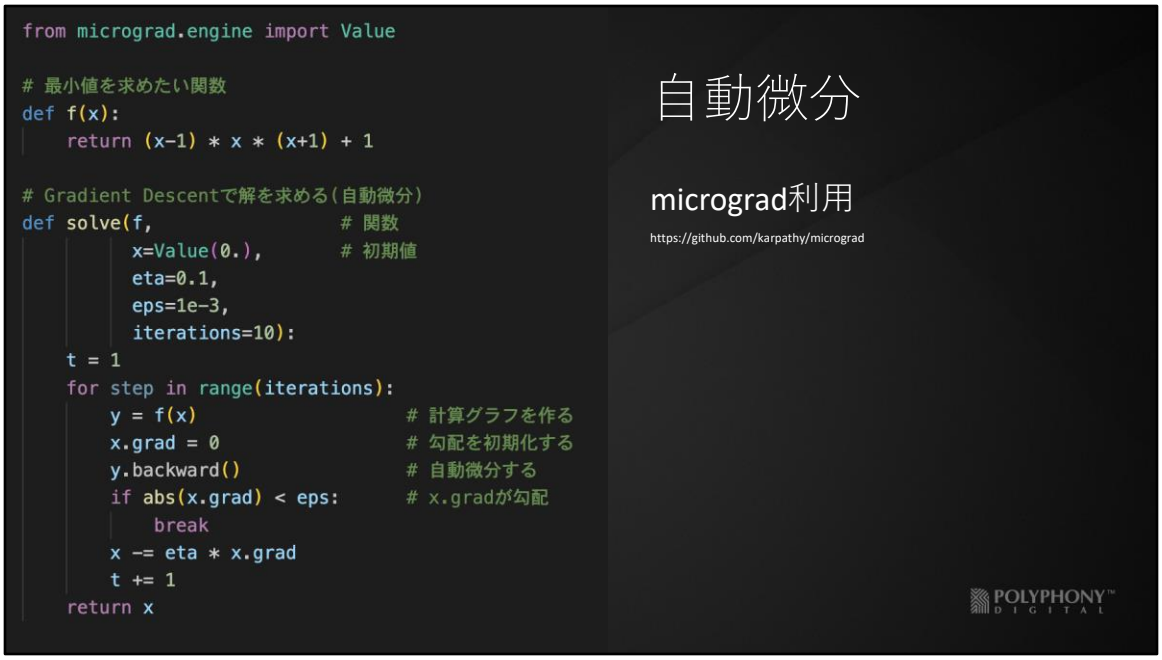
[Wikipedia:AD]

実際の計算では、裏側で計算グラフを記録していきます。

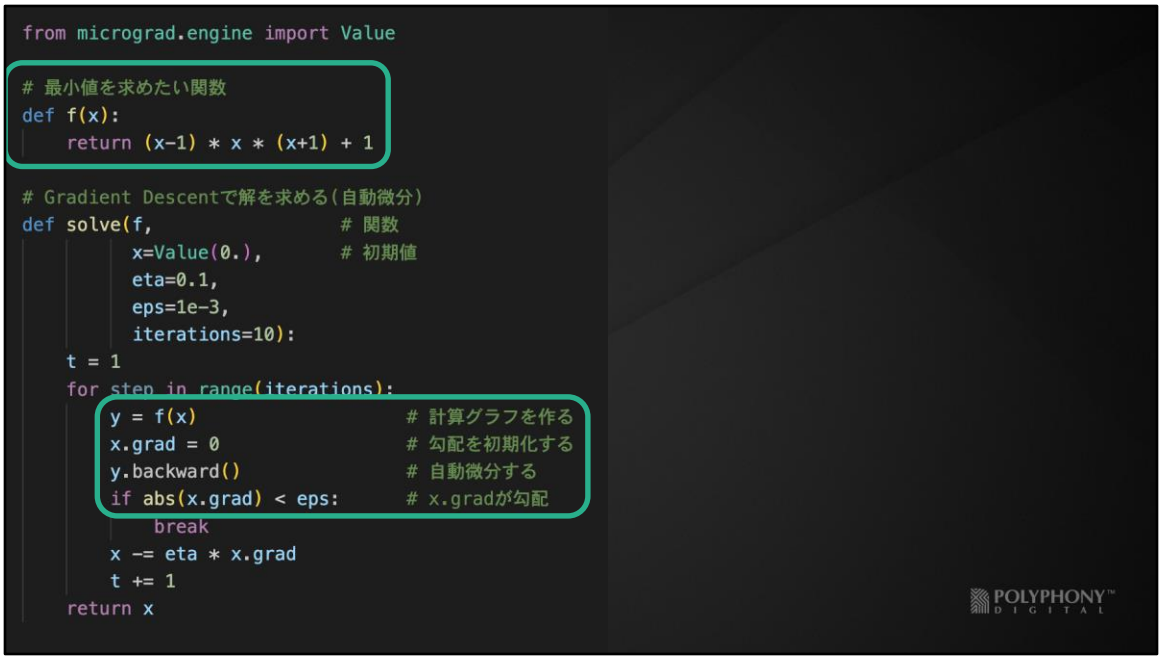
```python
class Var:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []
        self.grad = 0

    def __add__(self, other):
        return Var(self.value + other.value, [(1, self), (1, other)])

    def __mul__(self, other):
        return Var(self.value * other.value, [(other.value, self), (self.value, other)])

    def sin(self):
        return Var(math.sin(self.value), [(math.cos(self.value), self)])

    def calc_grad(self, grad=1):
        self.grad += grad
        for coef, child in self.children:
            child.calc_grad(grad * coef)
```

[Wikipedia:AD]

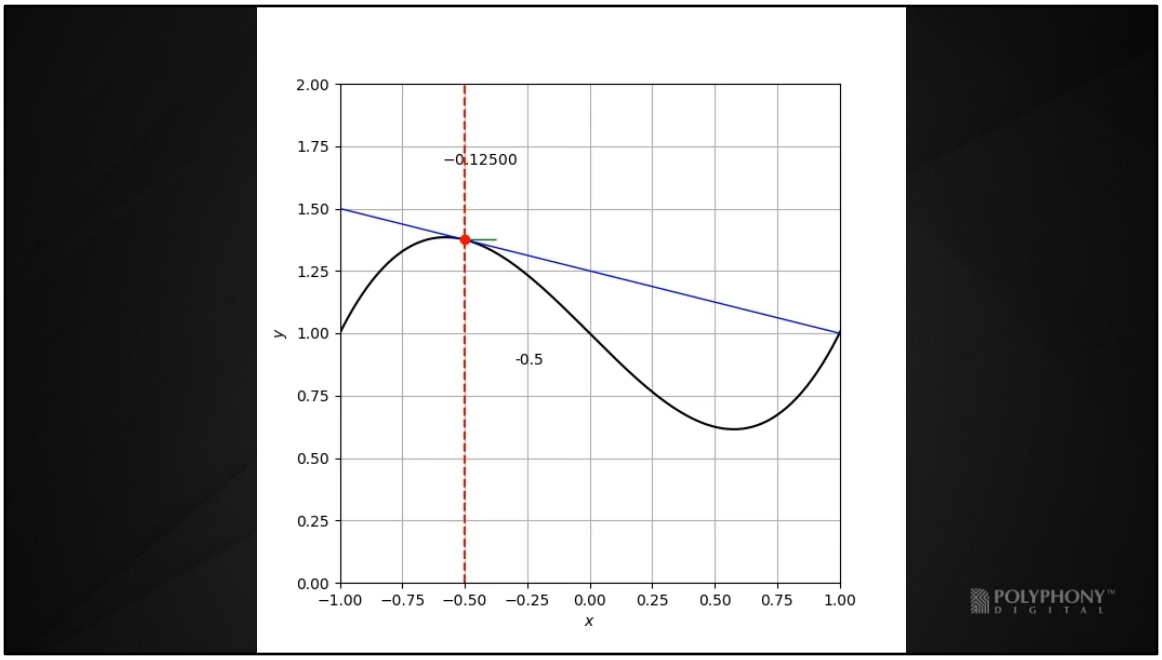
最後に、この計算グラフの記録をもとに、微分係数を求めます。実際の自動微分ライブラリはもっと効率的な計算や効率的なメモリ配置になっていますが、根本的な構造は同様です。



例えば自動微分ライブラリとしてmicrogradを使うと、先ほどのコードはこのようになります。

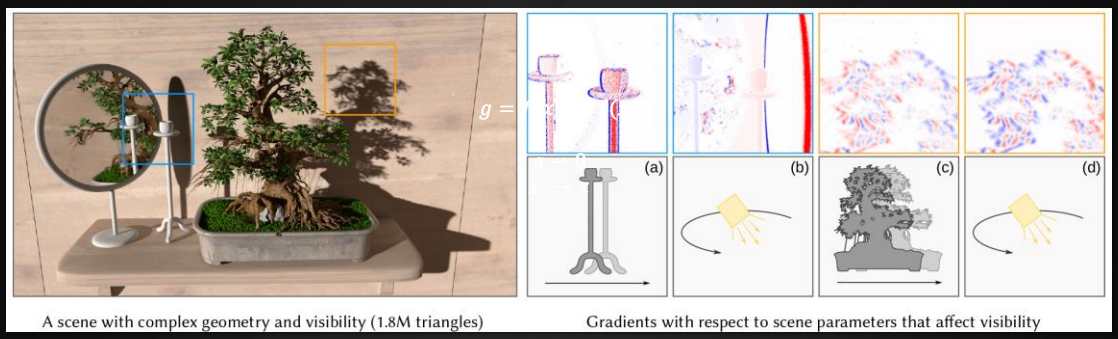


導関数を手で計算することなく、微分係数を得られます。
これが自動微分の利点です。



手で導関数を求めた場合と動作も変わりません。

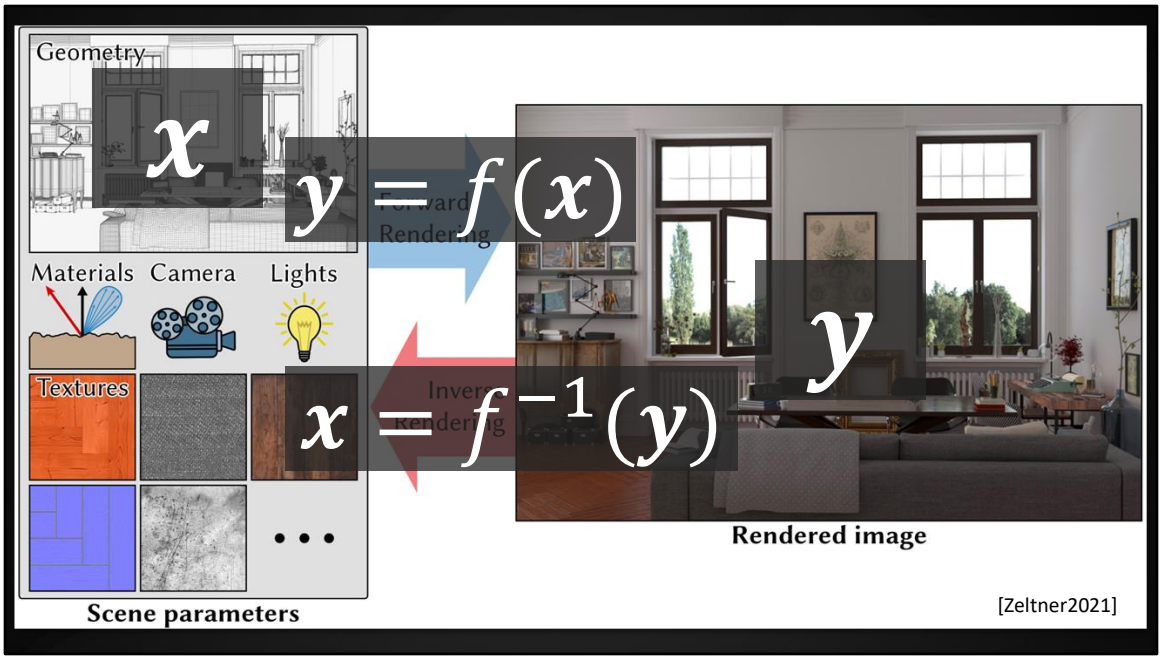
微分可能レンダリング



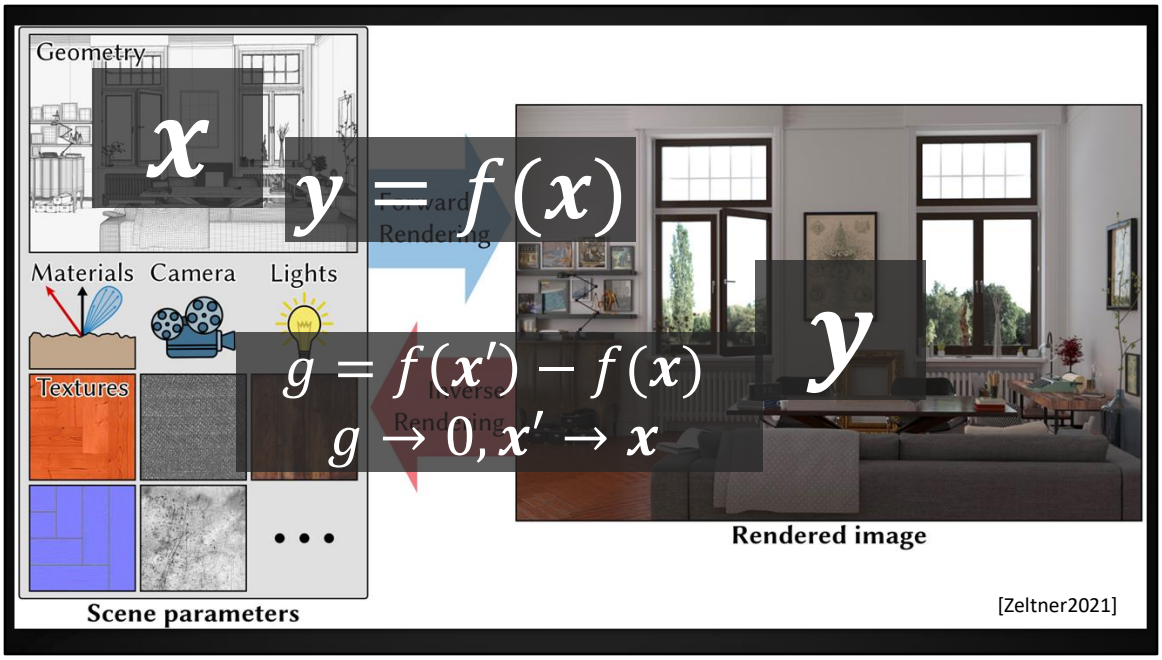
POLYPHONY DIGITAL

[Mitsuba3] [Loubet2019]

この自動微分の考え方を、レンダリングに関する諸問題にも応用したのが、
微分可能レンダリングと、微分可能レンダラーです。



さきほど、逆レンダリング問題を解きたい、という説明をしました。



$$\boldsymbol{x}$$

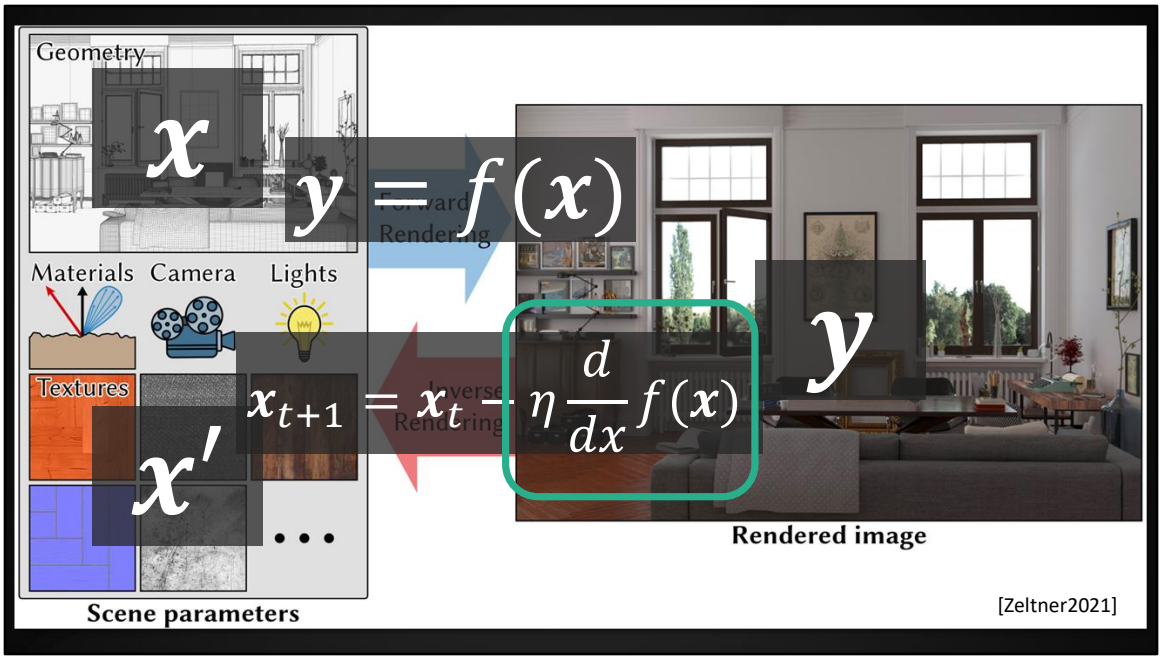
$$\boldsymbol{y} = f(\boldsymbol{x})$$

$$\boldsymbol{y}$$

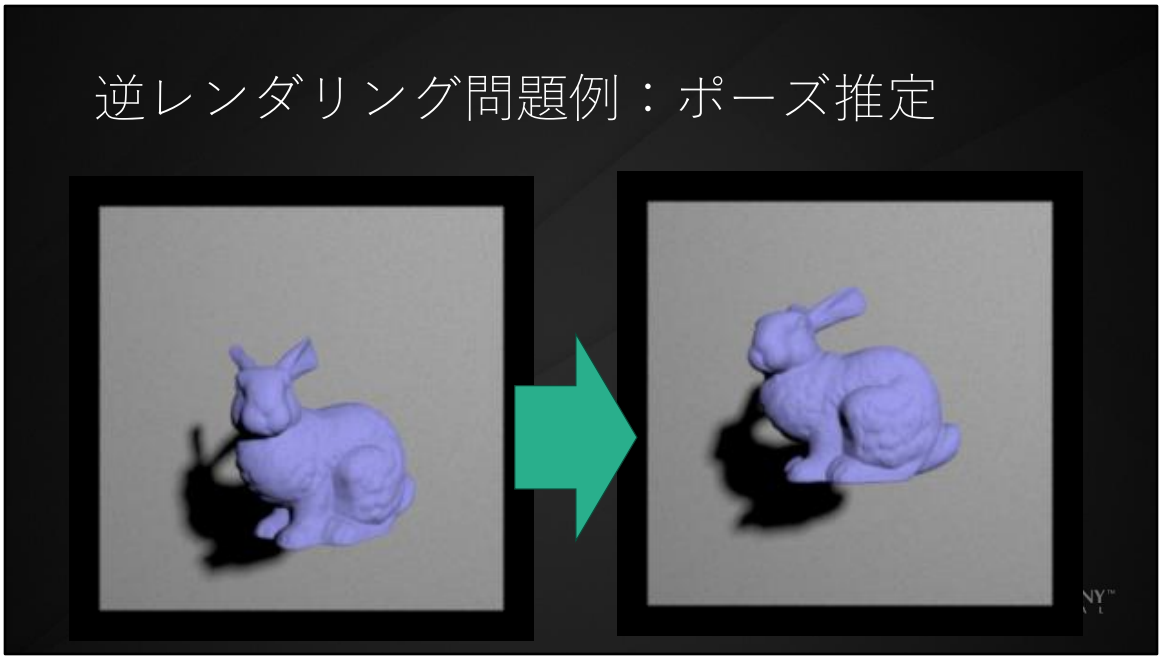
$$g = f(\boldsymbol{x}') - f(\boldsymbol{x})$$

$$g \to 0, \boldsymbol{x}' \to \boldsymbol{x}$$

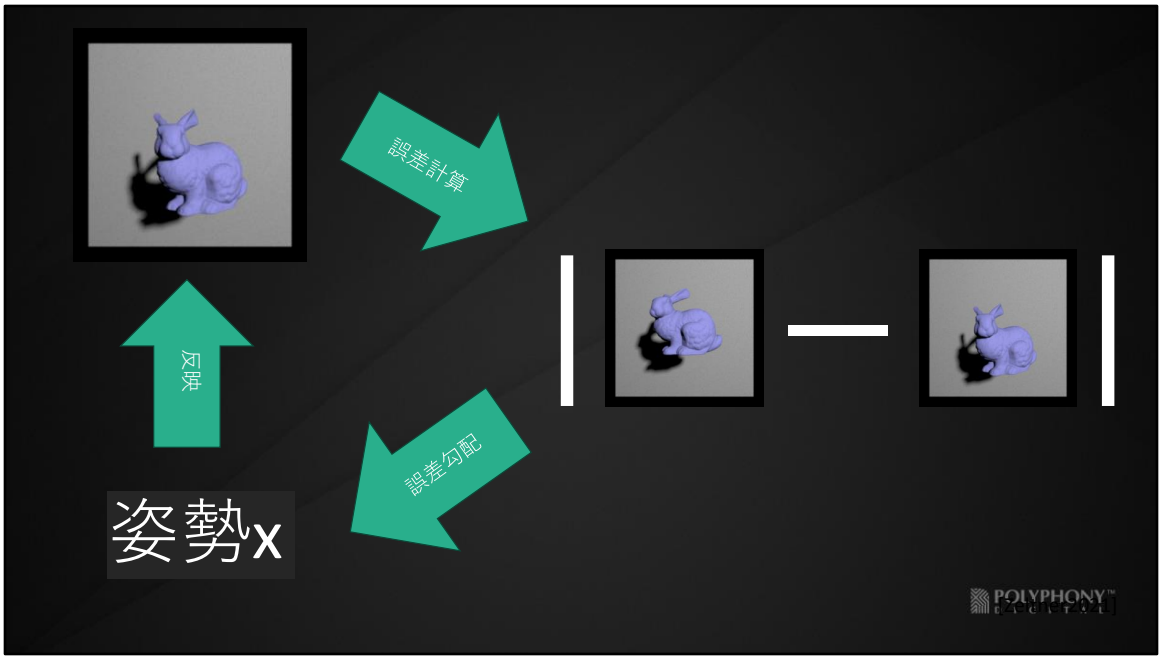
微分を使えば逆関数を求めずに解を求めることができるわけですから、



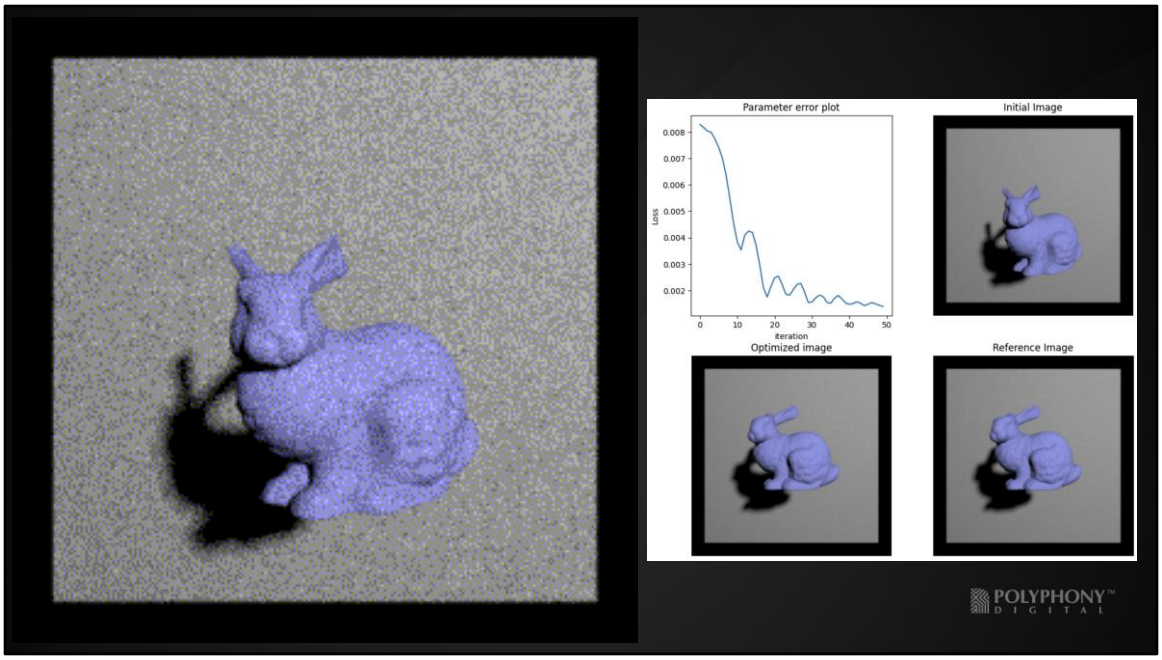
レンダリングが微分可能になれば、逆レンダリングをせずとも、解を求めることができるわけです。



微分可能レンダリングを用いることで、例えば、ポーズ推定をすることができます。
左のうさぎの姿勢が、右のウサギの姿勢になるように、姿勢パラメータ x を求めるような逆レンダリング問題を考えます。



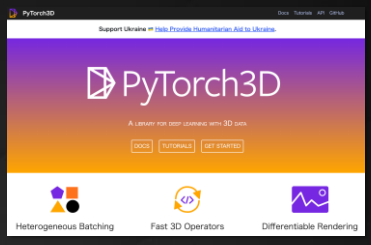
微分可能レンダラを用いれば、画像の差の微分係数を得ることができ、最小化問題として解くことができるのです。

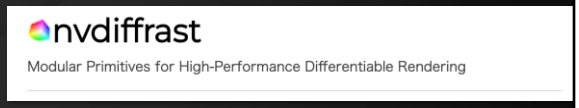


実際にこれを実装してみますと、このように、ポーズが推定され、誤差が最小化されていきます。

代表的な微分可能レンダラ

- PyTorch3D
  - https://pytorch3d.org/
- nvdiffrast
  - https://nvlabs.github.io/nvdiffrast/
- Mitsuba 3
  - https://www.mitsuba-renderer.org/





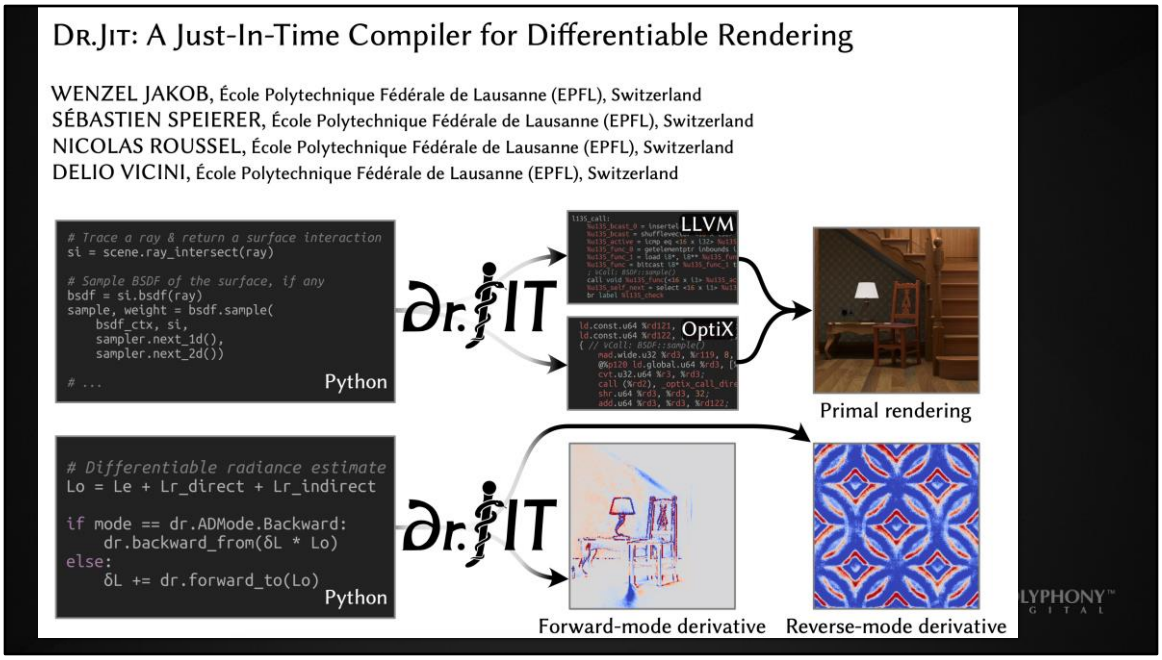
POLYPHONY DIGITAL

微分可能レンダラーの代表格としては、
これらの3つのレンダラが、あげられます。
PyTorch3DとnvdiffrastはラスタライザMitsuba3はレイトレーシング
という違いがあります。

今回、弊社ではMitsuba3をベースフレームワークとして用いることにしました。

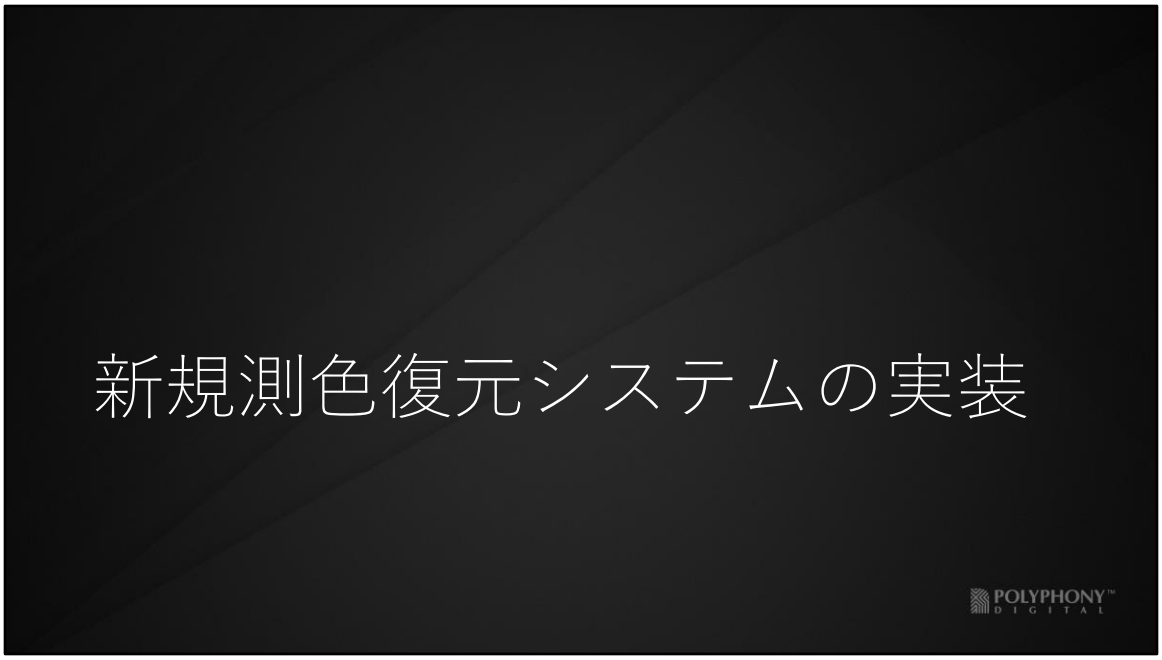
Mitsuba3は物理ベースの微分可能なレイトレーシングレンダラです。
Wenzel Jakob氏の研究室のメンバーらによって開発されており、
多くのテストで保証された、正確なレンダリング結果に定評があります。

Dr.JitというJust-In-Timeコンパイラバックエンドを内蔵していて、非常に高速な実行を特徴としています。
PythonバインディングでAPIが公開されていて、Pythonからの利用が非常に簡単です。

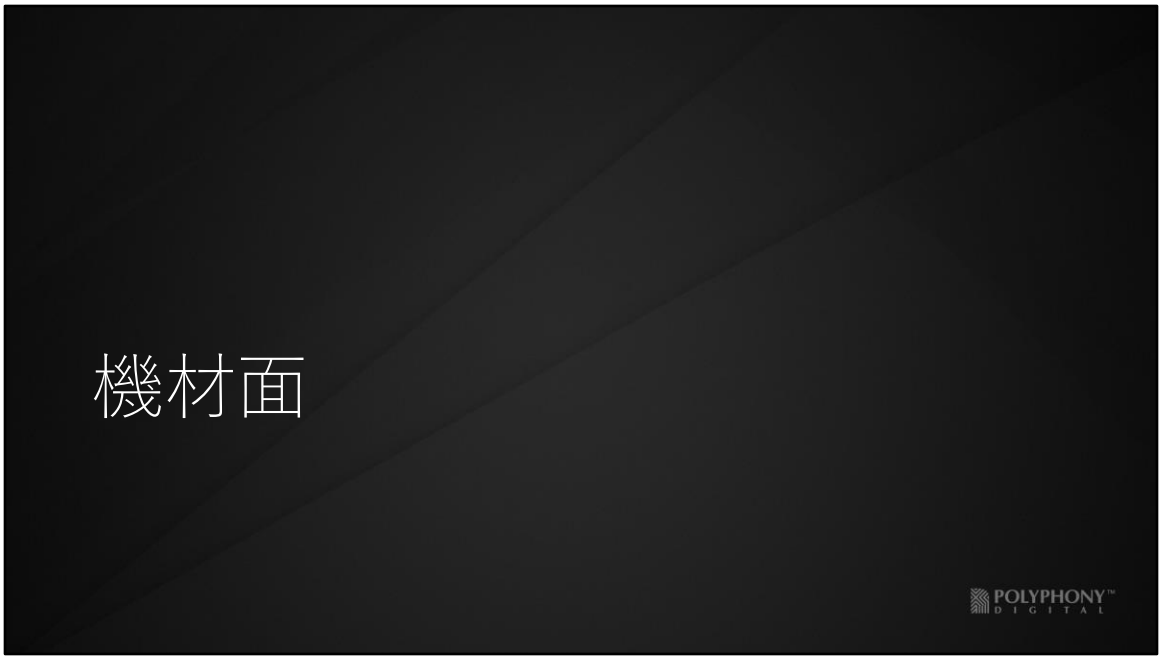


Dr.JitはMitsuba3のバックエンドで、
Pythonで実装されたさまざまなレンダリングエンジンやBRDFシェーダなどを
llvmやOptiX、cudaカーネルで高速に実行のできる、
微分可能演算コードを生成してくれるJustInTimeコンパイラです。
このコンパイラのおかげで、Pythonで実装したシェーダが自動微分されつつ
非常に高速に実行されて、
数値最適化を行うことができるのです。

新規測色復元システムの実装



ここまで説明してきた微分可能レンダラを用いて、実際の測色復元システムをどう実装したかについて説明します。



まず計測機器を更新しました。

BRDF計測装置を更新した

- X-Rite MA-T12
- Synopsys Mini-Diff v2

採用したのはX-RiteのMAT12と、SynopsysのMinidiff v2です。

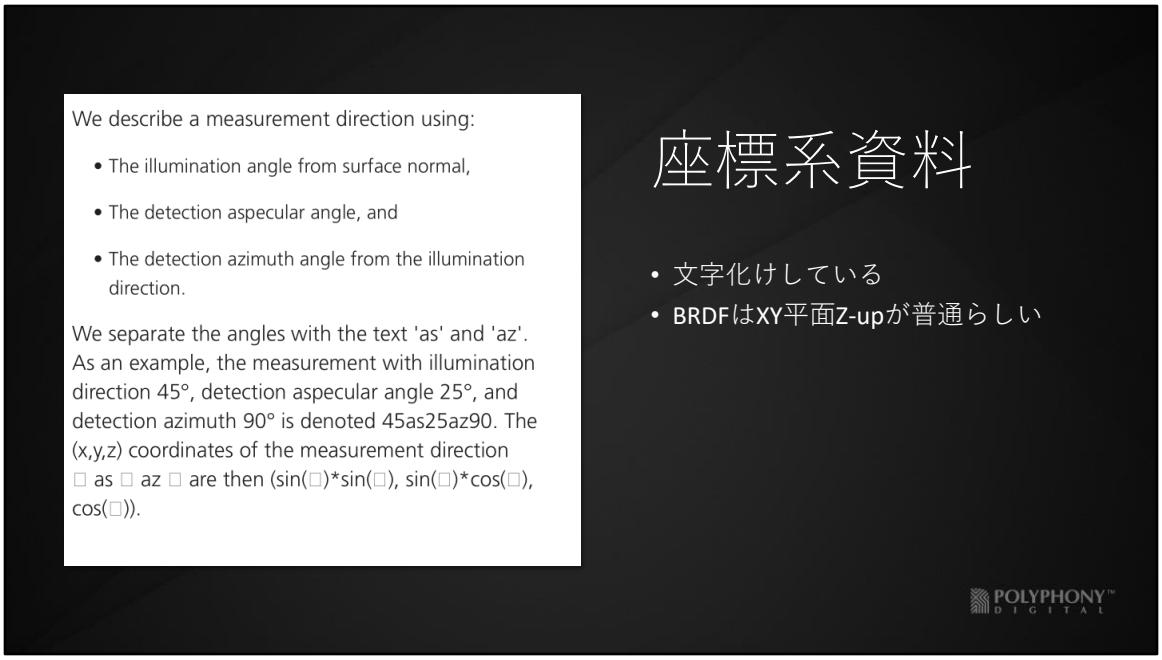
X-Rite MA-T12

- Pros
  - フルスペクトルの分光測色
  - 安定した計測
  - フレーク粒子の画像も撮れる
- Cons
  - サンプル角度が疎

[XRite:MAT12]

MAT12は、XRite社の多角度分光測色計です。
入射方向が2方向、出射方向が6方向、10nm刻みの分光での測色が可能です。
分光による精度の高い測色と、圧力センサでチェックされた安定した測定がメリットですが、
残念ながら入射方向、出射方向ともに非常に疎で、組み合わせでも12サンプルしか計測することができません。
その代わり、カメラセンサがあるので、フレーク粒子などを記録することができます。

MAT12の光学系の角度の資料はこのオフィシャルPDFが頼りなのですが、
実はこの資料には計算間違いがいくつか含まれており注意が必要です。



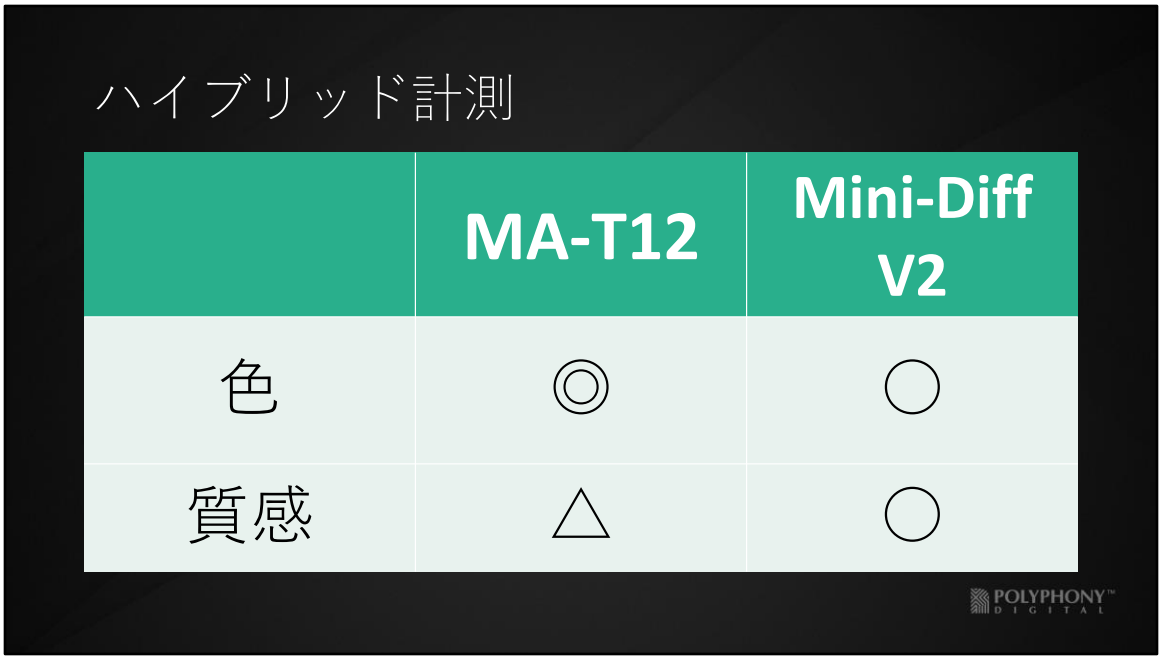
特に、一部の文字が完全に文字化けしており、読解には若干のエスパー力が必要とされます。

Synopsys Mini-Diff v2

- Pros
  - 1度刻みの高密度計測
  - 持ち運びが可能
- Cons
  - 色精度が微妙に不足
  - グレージングアングルは信頼できない

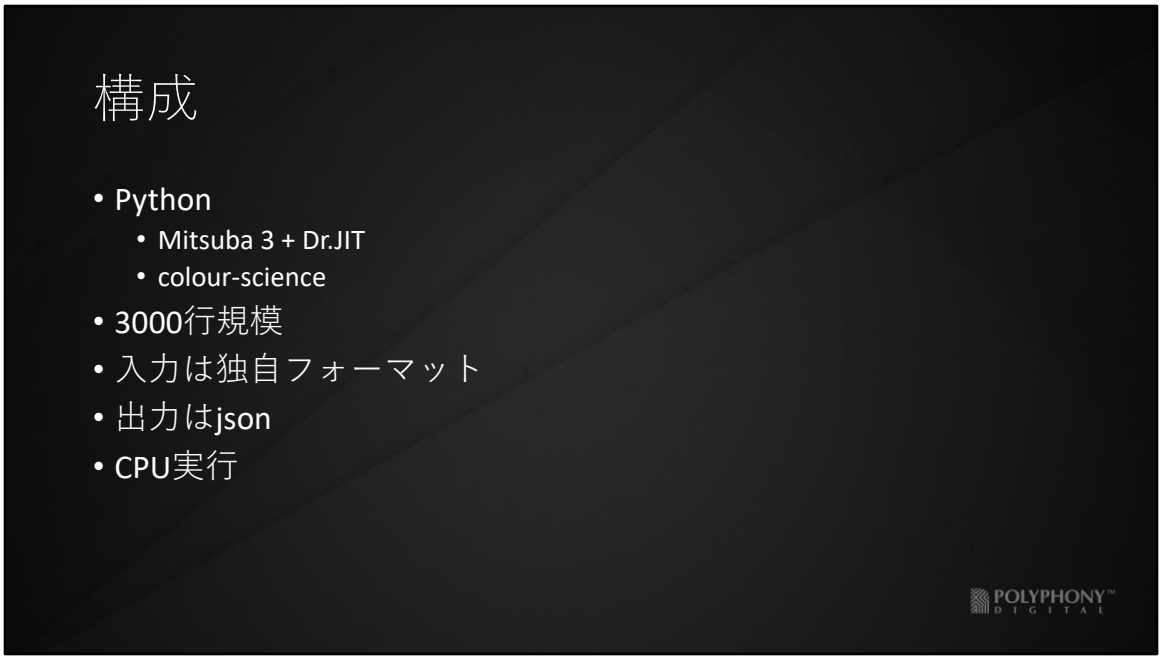
[Synopsys:Minidiff]

Minidiffは特殊なレンズとRGBのイメージセンサを用いて、
入射方向は4方向、出射方向は32400方向という
非常に密なサンプルを得ることができるBRDF計測装置です。
ただし、分光ではありませんので、色精度は高いとは言えません。
また、光学系の制約によって、グレージングアングル付近のサンプルは信用できません。
しかしBRDF計測装置としては非常に軽量かつ小型なので、ポータブルに持ち運びが可能です。



この二つの機材を組み合わせて、それぞれの特性を補う、ハイブリッド計測にすることにしました。
MAT12で計測した色情報によって、Minidiffのデータを色補正し、最終的な計測データとして扱います。

次に、これらの機材から出てきた計測データをどういったソフトウェアで処理したかを説明します。



大まかな構成はこのようになっています。Pythonでの実装で、CPU動作です。
行数はデータのローダなどを含めても3000行前後になっています。

大雑把なアルゴリズム

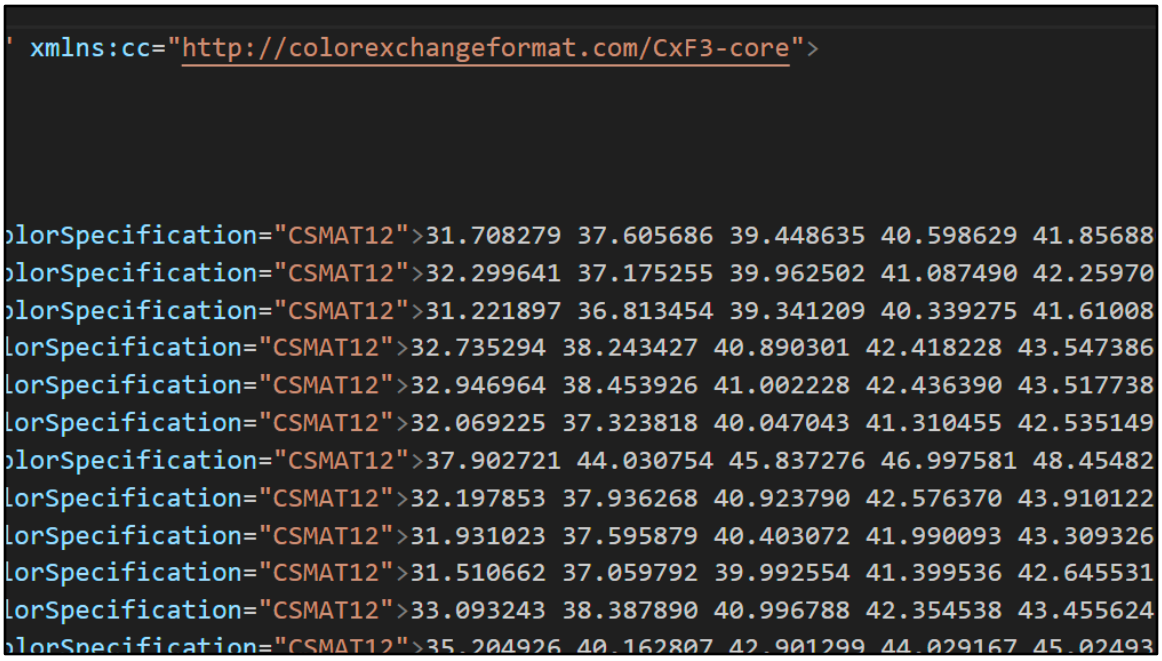
1. 計測BRDFデータをロード
2. 最適化対象BRDFを生成
3. 誤差を求める
4. パラメータを最適化する
5. 繰り返す

POLYPHONY DIGITAL

大雑把なアルゴリズムはこのようなステップになります。
計測データをロードし、
最適化対象のBRDFを生成し、
誤差を求め、
パラメータを最適化し、
収束するまで繰り返す。

```
LAB_NAME Light Tec

SYSTEM_NAME Mini-Diff V2
APERTURE Circular

SAMPLE_NAME BLUE_FLOWER

MEAS_NAME BRDF
FILE_NAME BLUE_FLOWER.astm
MEAS_DATE 2022/11/17
MEAS_TIME 19:41:23
NUM_POINTS 12964
VARS theta_i,phi_i,theta_s,465nm,525nm,630nm

0,3.14159265358979,0,0,0.123491747543674,0.0873476522974914,0.139365520337643
0,3.14159265358979,0.0174532925199433,0,0.129129540972326,0.0893321032447395,0.1335814183
0,3.14159265358979,0.0349065850398866,0,0.133694190171798,0.0883066641977171,0.1278781272
0,3.14159265358979,0.0523598775598299,0,0.141221537363785,0.0932413266387739,0.1394697509
0,3.14159265358979,0.0698131700797732,0,0.141789583138289,0.0918856376858154,0.1280149014
0,3.14159265358979,0.0872664625997165,0,0.137599633353517,0.0887833496909198,0.1246535058
0,3.14159265358979,0.10471975511966,0,0.132325965338796,0.0857679604966609,0.121446925985
```

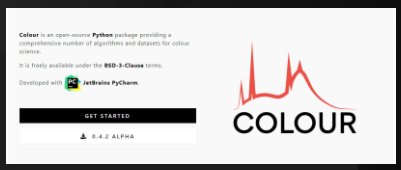
まず、計測データをロードします。
それぞれの機器から出力されてくるのは、
入射方向と、出射方向、そして反射率の羅列です。



ただしMAT12は分光反射率でデータが出てきますので、
これを等色関数を用いてRGBに変換してから利用する必要があります。

色彩工学は
colour-scienceライブラリを利用

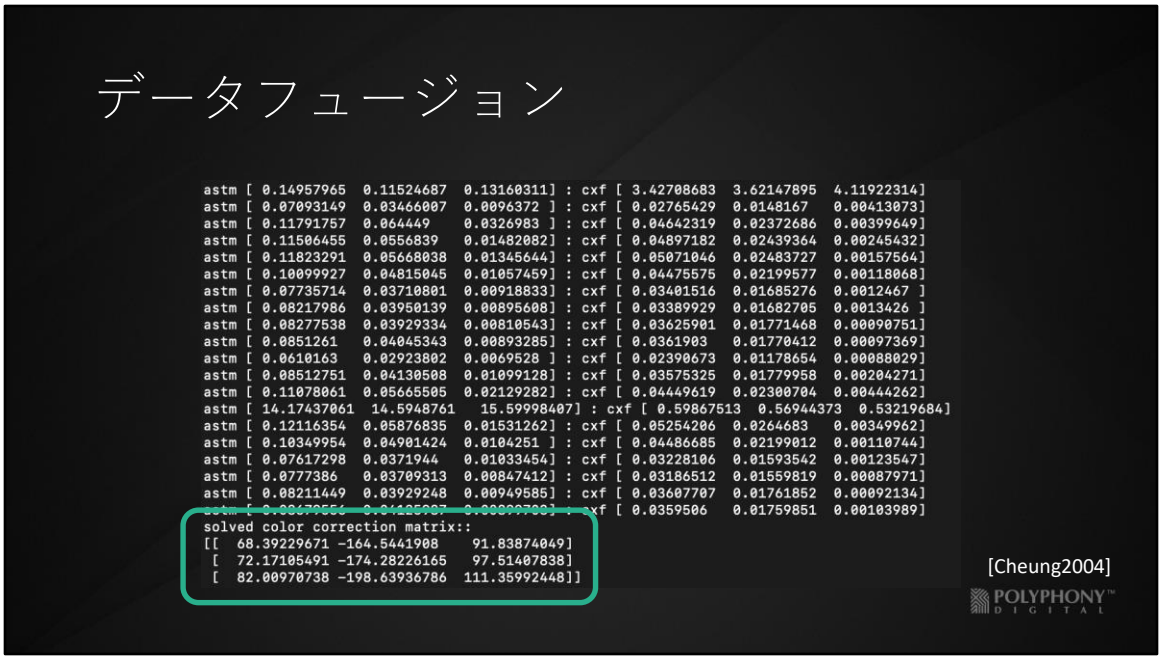
```
def set(self, xyz, wi, wo, norm=[0, 1, 0], angleText=''):
    self.angleText = angleText
    self.weight = 1
    self.xyz = xyz
    self.xyY = colour.XYZ_to_xyY(xyz)
    self.rgb2020 = colour.XYZ_to_RGB(
        xyz, None, None, colour.RGB_COLOURSPACES['ITU-R BT.2020'].matrix_XYZ_to_RGB, None, None)
    self.wi = wi
    self.wo = wo
    self.sph_i = brdfCoords.vec2sph(wi)
    self.sph_o = brdfCoords.vec2sph(wo)
    self.wAspecular = brdfCoords.reflect(wi, np.array(norm))
```


Colour is an open-source Python package providing a comprehensive number of algorithms and datasets for colour science.
It is freely available under the BSD-3-Clause terms.
Developed with JetBrains PyCharm.
GET STARTED
0.4.2 ALPHA
COLOUR

https://www.colour-science.org/

POLYPHONY DIGITAL

RGBへの変換にはcolour-scienceという色彩工学ライブラリを利用しました。



二つの計測器のデータをフュージョンして扱っているわけですが、実際には色補正行列を求めて事前スケーリングをかけています。
Minidiffのデータと、MAT12のデータから、同じ方向のサンプルを抽出し、Cheungらによる2004年の手法を用いて色補正行列を求めています。

次に、最適化対象のBRDFオブジェクトを生成します。
Mitsuba3では、シーンに対して最適化を行うので、シーン内に明示的にBRDFオブジェクトを生成する必要があります。
Mitsuba3は組み込みのBRDFをいくつも持っていて、非常に簡単に生成し利用することができます。
Pythonの辞書型にキーワードとパラメータなどを設定して、loadするとそのまま使うことができます。

ここでは、PrincpledBRDFを生成していますが、他のBRDFを最適化する場合もほとんど同様に書くことができます。
その場合は、ここの type キーワードを用います。

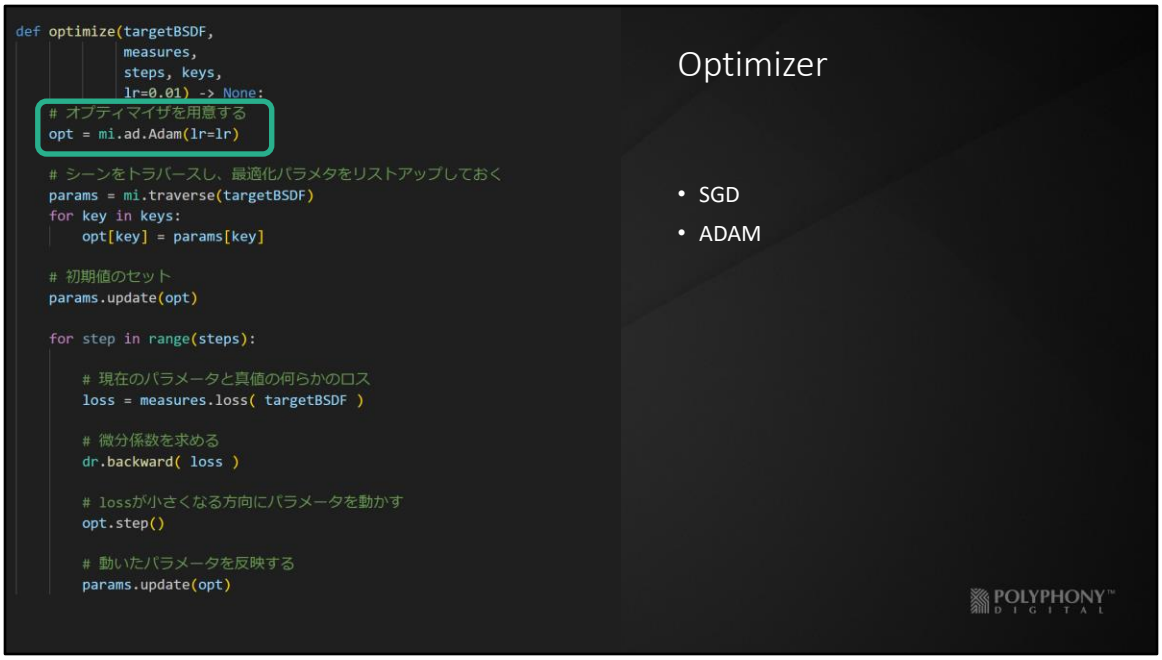
同時に、最適化したいパラメータ名なども保存しておきます。

組み込みではないBRDFも、Pythonプラグインの形で自由に実装することができます。
しかも、Pythonで書かれているにもかかわらず、バックエンドのDr.JITがこれをLLVMバイナリやCUDAバイナリにコンパイルしてくれるので、非常に高速に実行されますし、いくつかの制約を守れば自動微分もそのまま利用可能です。
テクスチャのルックアップなども微分されます。
この仕組みを使って、Principled以外のBRDFのパラメータを求めることも可能になりました。

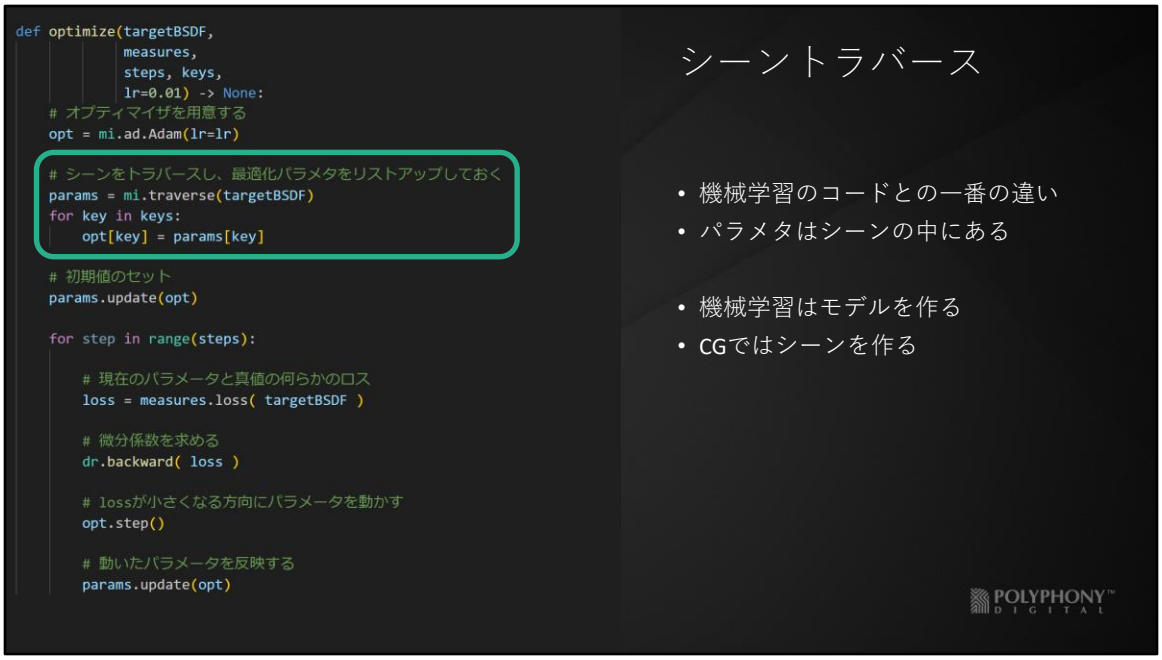
こうしてロードした計測データとBRDFに対して、最適化ループを回します。
そうです。BRDFフィッティングの実態は、たったこれだけで書けてしまうのです。

もちろんこれは抜粋してかなり簡略化した最適化ループですが、
基本的にはこれだけのコードで、マテリアルのパラメータ推定ができてしまいます。
過去の測色復元システムで複雑な10000行のC++をメンテナンスしていたのに比べると圧倒的です。

最適化途中はこのようにパラメータが変化していきます。



簡単にステップを追ってみましょう。
まず一番最初に、mitsuba3組み込みのオプティマイザを用意します。
SGDとADAMを利用することができます。



次に、シーンをトラバースし、最適化するパラメータをリストアップします。
ここが機械学習フレームワークと微分可能レンダリングの大きな違いです。
機械学習ではモデル全体を最適化します。
計算グラフは、ロス関数を計算するときに決定します。

微分可能レンダリングではこの順序が逆で、
シーンを事前にトラバースすることによって、
レンダリングに関係のない計算グラフを検出したり最適化を行います。

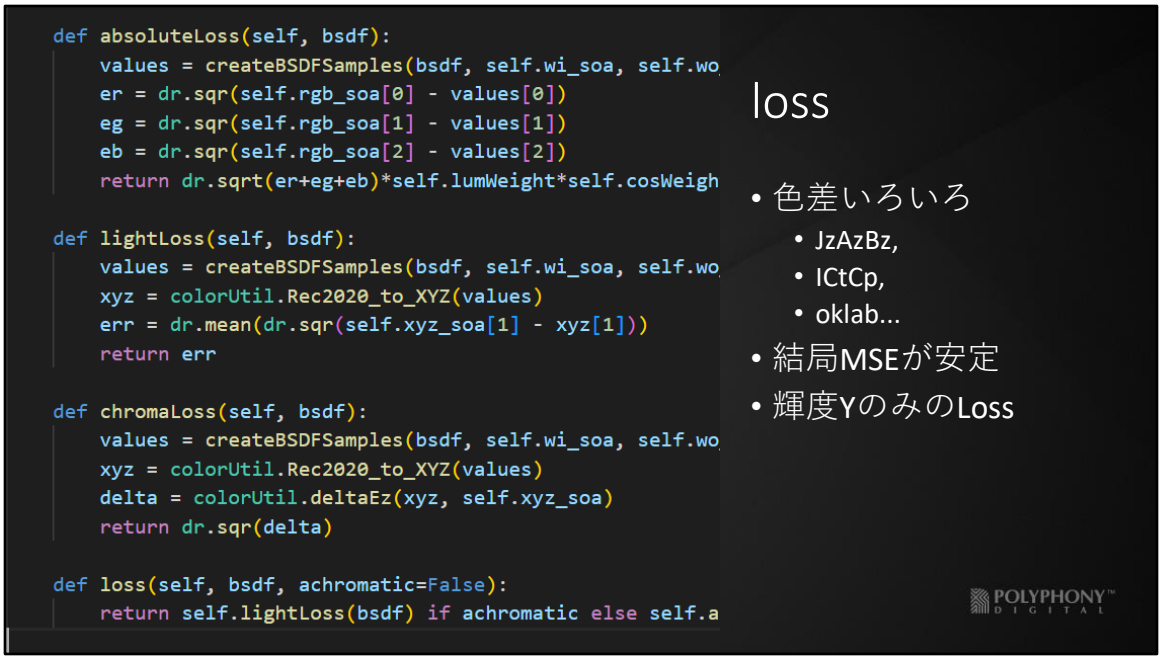
ここから最適化ステップで、真値とのロスを求めます。
誤差とロスとの違いについて説明します。
誤差は真値と予測値の差です。
ロスは、とにかく小さくできたら嬉しいなんか謎の値、です。
つまり、誤差をロスとしてもいいですし、何か全く違うものをロスとしても良いのですが、
Lossを誤差と呼ぶと場合によって数学警察に怒られます。
この、lossこそが、最適化の肝になります。

def optimize(targetBSDF,
measures,
steps, keys,
lr=0.01) -> None:
# オプティマイザを用意する
opt = mi.ad.Adam(lr=lr)
# シーンをトラバースし、最適化パラメタをリストアップしておく
params = mi.traverse(targetBSDF)
for key in keys:
opt[key] = params[key]
# 初期値のセット
params.update(opt)
for step in range(steps):
# 現在のパラメータと真値の何らかのロス
loss = measures.loss(targetBSDF)
# 微分係数を求める
dr.backward(loss)
# lossが小さくなる方向にパラメータを動かす
opt.step()
# 動いたパラメータを反映する
params.update(opt)
自動微分する
• lossを最小化する勾配を求める
POLYPHONY DIGITAL

微分係数を求め、

パラメータを更新します。

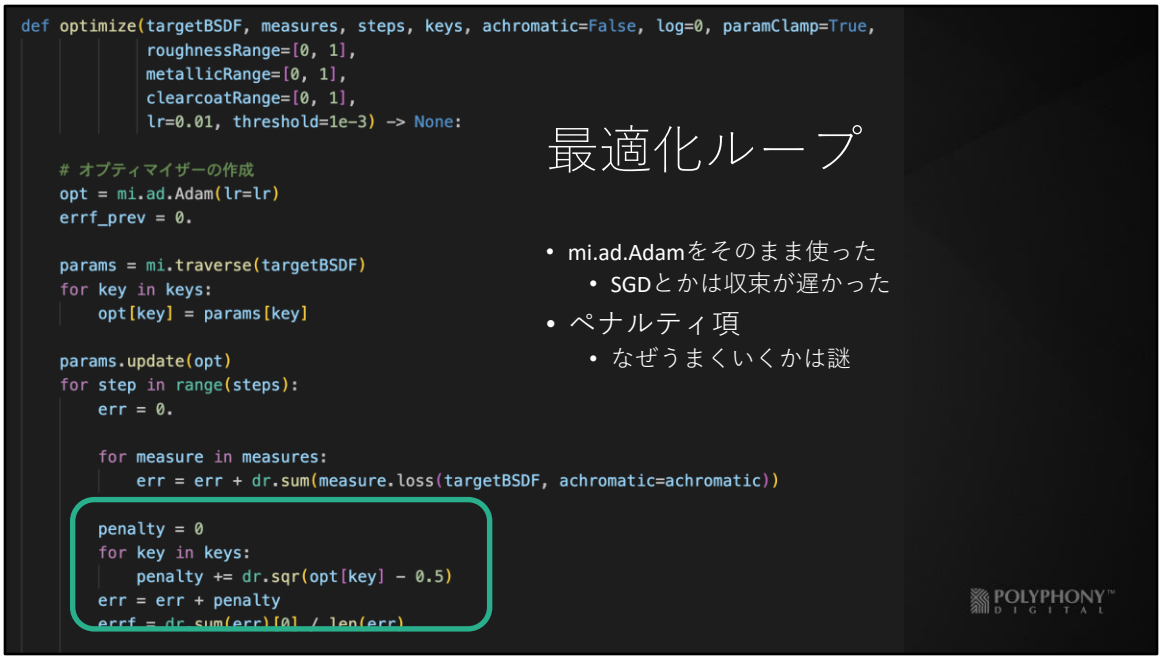
つまり、最適化ループのほとんどは、loss関数の出来に依存しています。



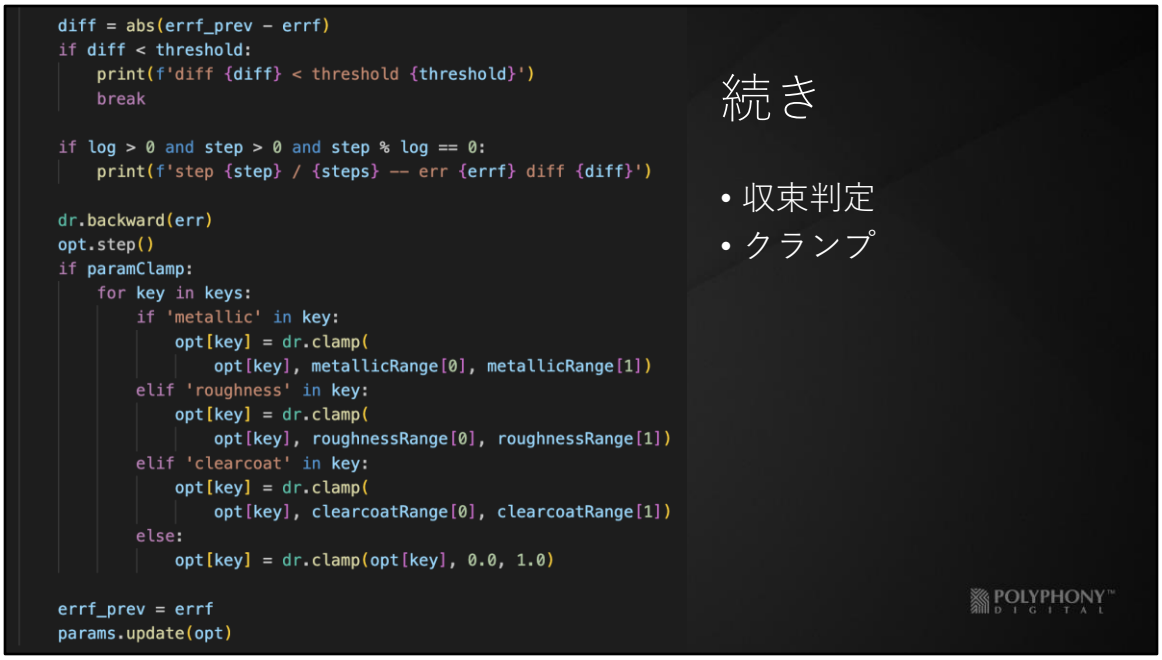
とはいえloss関数自体もMitsuba3の機能を用いれば非常にシンプルに書くことができます。
計測データのそれぞれの入出方向でBRDFの値を求め、単純な二乗誤差を求めています。
さまざまな色空間や色差を試したりしたのですが、結局、単純な手法が最も安定していい結果を出しました。後述する二段階最適化のために、輝度のみの差を考えるロス関数も利用しています。

今回、二つの計測器のデータをフュージョンして利用していますが、
事前に色補正のためのスケーリングをしていることもあり、
単純な合算でロスとしています。

ロス計算に用いるBRDFサンプリングも、組み込み関数が用意されており、非常に簡単です。
入射ベクトルと出射ベクトルのリストを用意して、Mitsuba3の組み込み関数を呼ぶだけで
高速にサンプリング結果のリストが得られます。

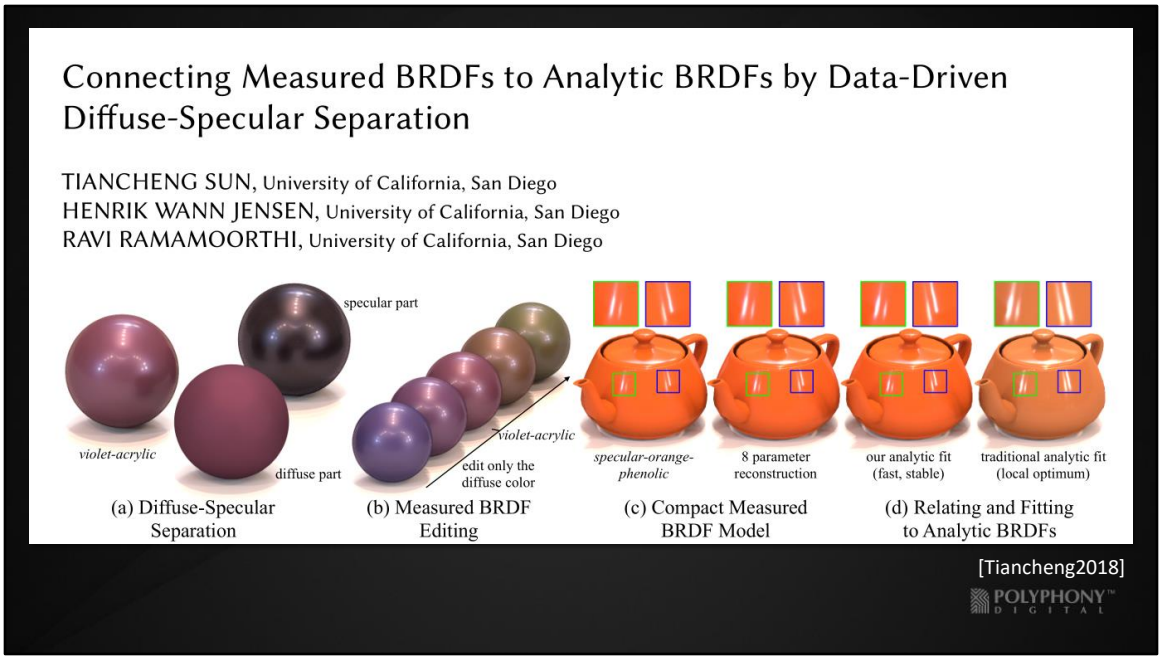


実際のコード上の最適化ループはもう少し複雑です。
最適化手法としては、Adamを用いています。
SGDなどは収束が遅く、あまり良くありません。
ペナルティ項にはパラメータが偏るのを防ぐ効果があり、重要です。
とはいえ、うまくいっている理由は実はあまりわかっていません。
ないよりもあったほうがうまくいくので、入っています。



最適化ループの中ではロスの値の変動を見て、収束判定をしています。
収束判定は非常に難問で、今のところこれが正解！と言えるコードは書けていません。
パラメータクランプも重要です。
Principled　BRDFはパラメータの値域が0-1です。
パラメータをクランプすると非連続面になってしまいますので、あまり望ましくありません。
学習率を上げるとクランプ面に高速に突っ込むらしく、発散しやすくなります。
ここは何か変数変換などを使ってもう少し上手に解決できる手法がありそうです。

この最適化ループの一段外側に、別の最適化を用意しています。
BRDF関数はローブ形状が非常に急峻なため、単純な色差では局所解に陥ることがあるため、
二段階に最適化を行い、初期値を安定させています。
データセットはそのまま、ロス関数を切り替えるだけで実装が可能です。



この二段階最適化はUCサンディエゴのTiancheng Sun氏らによる、
論文のアイデアをもとにしています。

二段階最適化

- 最初に輝度のみを用いて
  - Metallic
  - Roughness
  - Clearcoat
- 次に3要素MSEで
  - Basecolor
  - Metallic
  - Roughness
  - Clearcoat

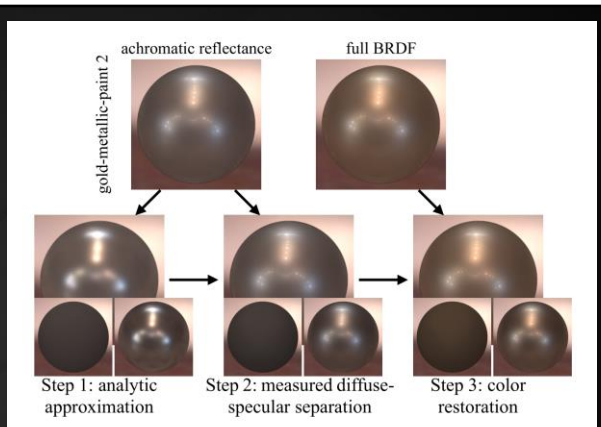
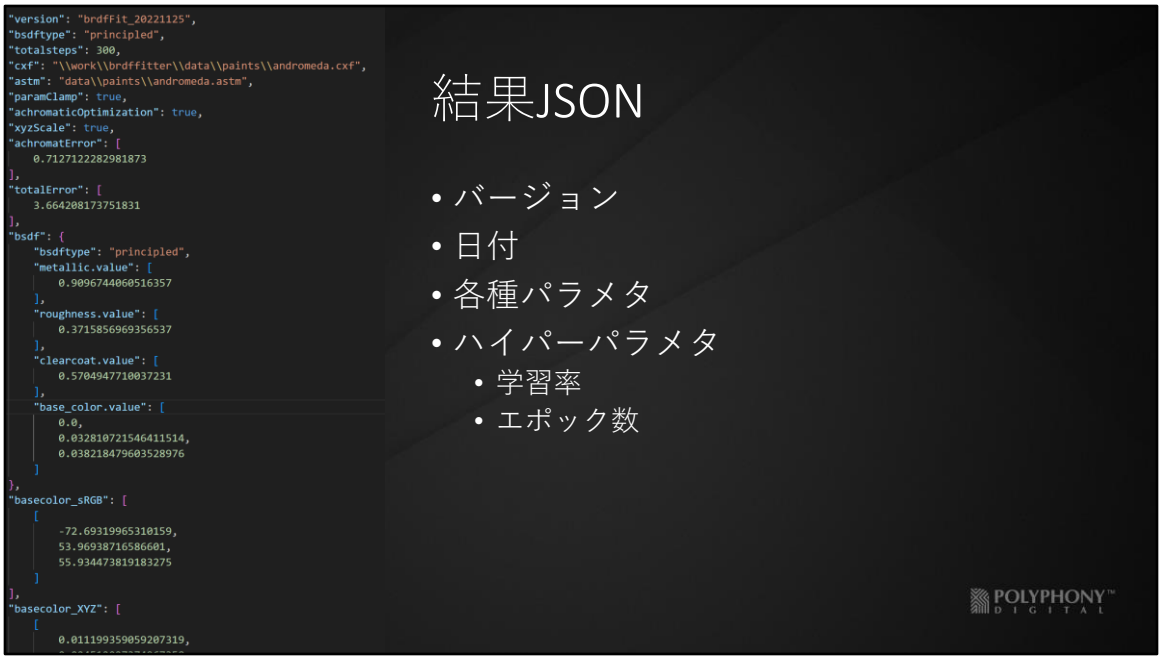


Fig. 2. An overview of our diffuse-specular separation algorithm using 3-step optimization. We first fit an analytic BRDF to the *achromatic reflectance* (average reflectance across color channels) of the target measured BRDF. Then, we use the analytic BRDF as guidance to separate the diffuse and specular parts of the measured BRDF. Finally, we restore the colors for each part. Note that since we render the BRDFs under a full environment map, the images may have some color even though the BRDFs are achromatic in steps 1 and 2.

[Tiancheng2018]

具体的には、
最初は輝度のみの差でMetallic Roughness Clearcoatを最適化します。
次に、RGBを考慮した色差で、Basecolor も含めた最適化を行います。
同様のテクニックは複数の論文でも採用されており、
解析的BRDFモデルにおける局所解を回避する手法として有用です。



結果はJSONファイルに出力し、後段のDCCツールにパラメータを渡しています。
今までの経験上必要そうなメタデータはできるだけ多めに保存しておくようにしています。
例えば、バージョンや実行した日付、元データのファイル名などを保存しておくと、
ツールを更新した際に特定ファイルだけ再実行するなどの特殊対応がしやすくなります。

結果表示も超簡単

```
targetBSDF_params.update()
scene = mi.load_dict(matpreview.matpreviewScene(targetBSDF, spp=128))
image = mi.render(scene)
figure = plt.figure()
subplt = figure.add_subplot()
subplt.axis("off")
subplt.imshow(image ** (1.0 / 2.2))  # approximate sRGB tonemapping

if args.savepreview:
    mi.util.write_bitmap(args.savepreview, image)
```

POLYPHONY DIGITAL

レンダリングはそもそもmitsuba3の本分ですから非常に簡単です。
最適化されたパラメータをシェーダに入れて、シーンを構築しrenderするだけで
美麗なレンダリング結果を得ることができます。

今回はマテリアルパラメータの評価用ですので、マテリアルボールにBRDFをアサインしてレンダリングするようにしました。

処理時間

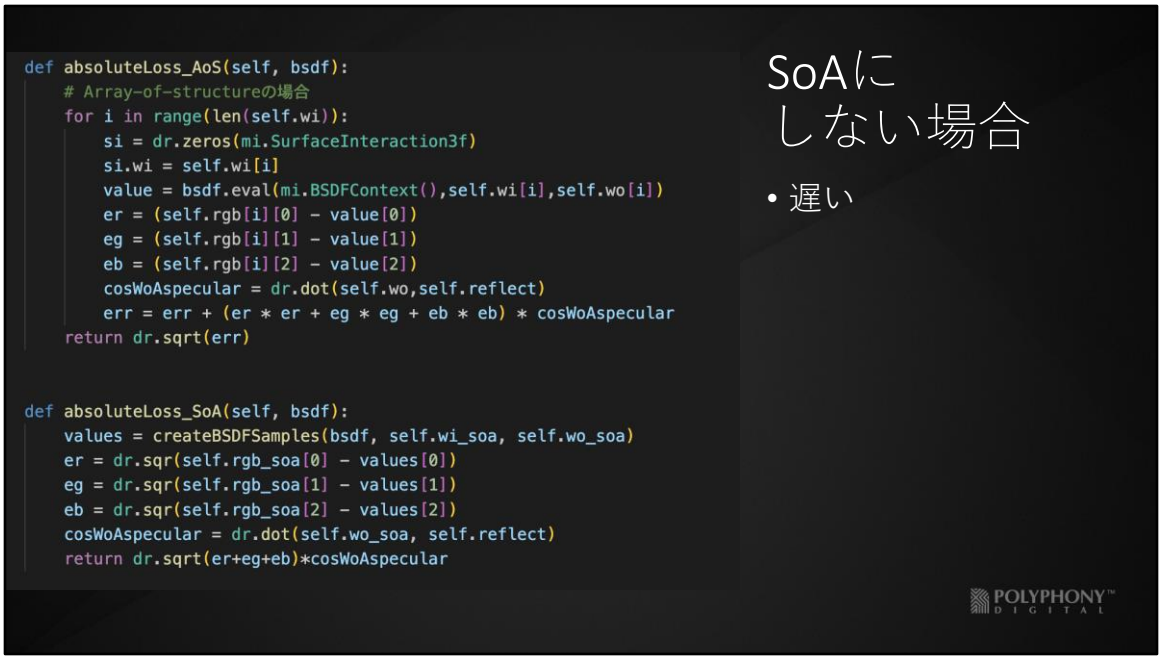
```
20.66s user 1.48s system 120% cpu 18.393 total
```

- M1 Max MBPでおよそ20秒前後
- Python実装としては十分に速い
  - Dr.JITありがとう
- 結果のレンダリング画像を追加するともう少しかかる

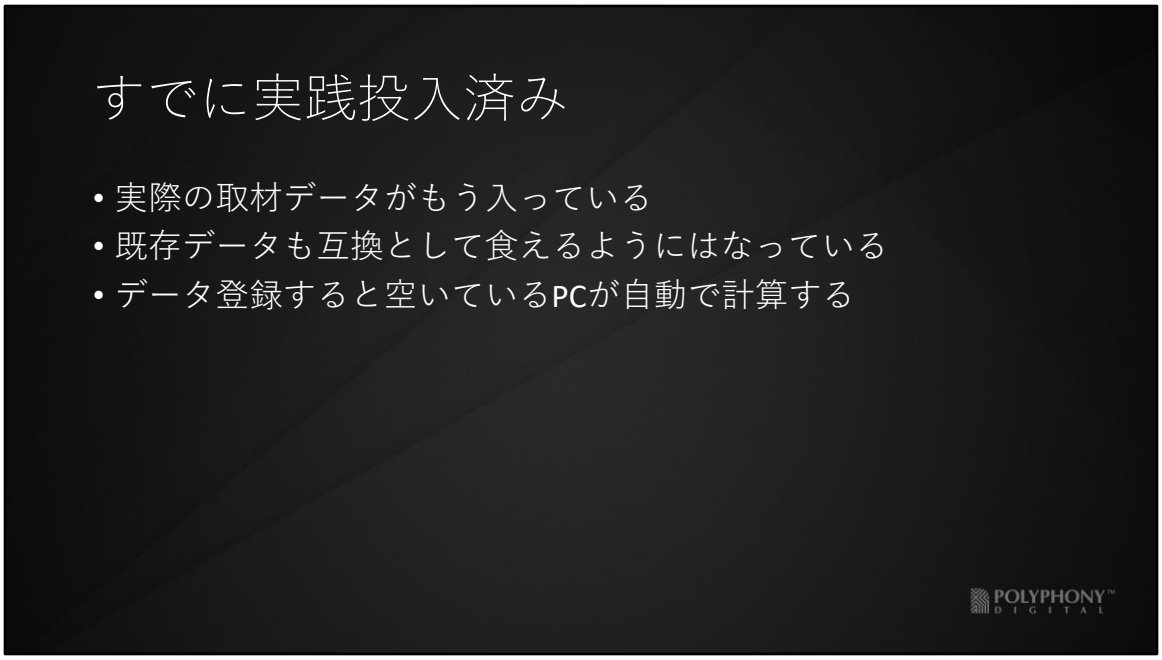
POLYPHONY DIGITAL

処理時間は収束しやすいデータかどうかで変わりますが、手元のノートPCでおよそ20秒から1分前後です。
ほぼ全てPythonで実装されているのにこのスピードで動いているのは、Dr.Jitによる恩恵です。

少し余談になりますが、Dr.JITの恩恵を最大限に受けるためには、Structure of arrayを使います。
Numpyでの演算に慣れている方は自然にやられているかもしれません。
普通にファイルからロードしてきたデータなどはArray of structureになることが多いかと思います。
演算の前にunravel関数できちんと並び替えをしてやると、長いSIMDベクトルのように演算され、とても高速になります。
C++歴が長かったので、この書き方に慣れるまでに苦労しました。



たとえばロス関数をSoAを使わずに書くとこのようなコードになり、せっかくのDr.JITによる高速化の恩恵が受けられなくなります。
速度比較をしようと思ったのですが、ちょっとやる気が出ないくらい遅かったので、具体的な数値はありません。

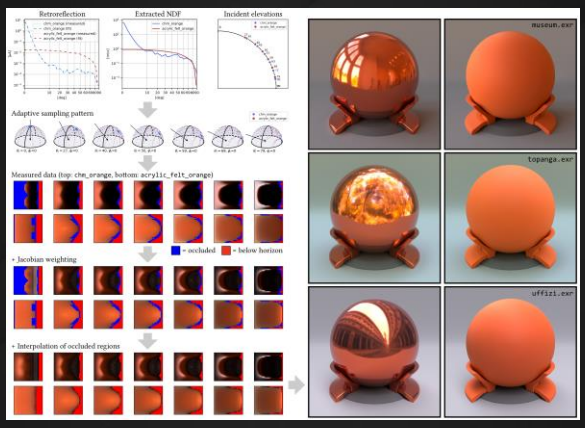


こうして作成した新測色復元システムは、もう実戦に投入されていまして、
すでに出荷済みのアセットにもこのデータが入っています。
アーティストが取材で計測してきたデータを社内のデータベースに登録すると、
CPUの空いているPCが自動でアサインされて、最適化計算が走る仕組みになっています。

復元サンプル紹介

ここからは、作成した測色復元システムの実際のサンプルを紹介していきます

RGL EPFLデータベースの紹介

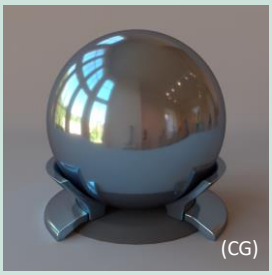
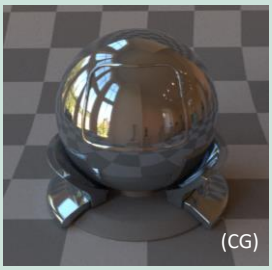


[MERLBRDF]
[RGL:MaterialDB]

今回は、信頼に足る真値として、RGLEPFLのマテリアルデータベースを利用しました。
これはWenzel Jakob氏らによる実在マテリアルの計測BRDFのデータベースです。
計測BRDFデータのみならず、そのデータを用いてレンダリングした画像も添付されているため、
比較的、評価用画像として利用しやすいものになります。

同様のBRDFデータベースとして三菱電機のMERLデータベースが知られていますが、
いくつか既知の問題があったため、今回は利用しません。

RGL EPFLとの比較

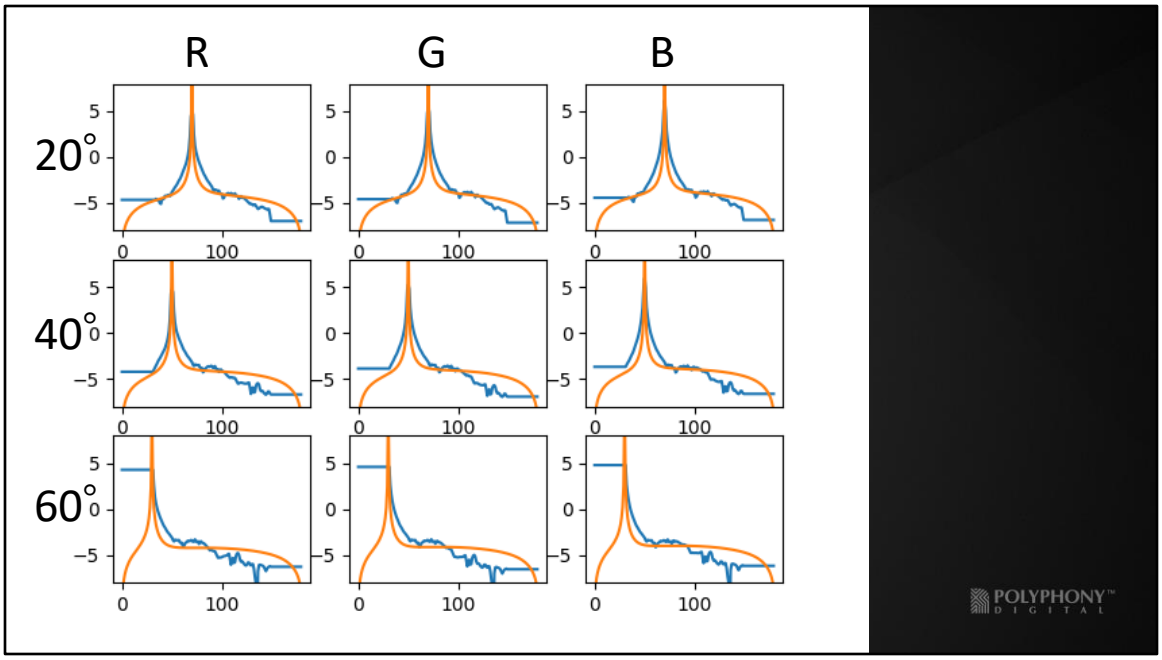
chm_light_blue	新システム	旧システム
 (CG)	 (CG)	(CG)
Metallic	0.8655744194984436	0.87732505798339844
Roughness	0.0	0.46252202987670898
Clearcoat	1.0	0
Basecolor	149/164/176	25/25/32

まずは一般的な金属質のマテリアルから復元してみます。
旧システムは明らかに推定に失敗しています。

新システムは旧システムとは逆に、過剰にスムースに見えますが、
全体としての再現度は新システムの方が良く見えます。

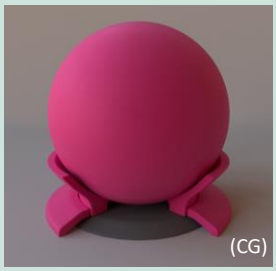
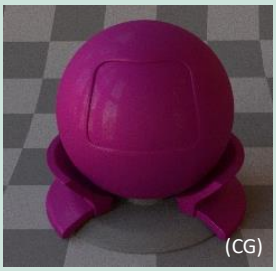
旧システムはこのように失敗してしまったときが大変で、なぜ失敗したのか？を
大量のログをもとに追い、ハイパーパラメタを調整して対応する必要がありました。
今回のシステムは可視化などデバッグ手段がより増えたため、デバッグ対応が少し楽になるといいなと期待しています。

参考モデルも完全に同一のレンダリング条件でレンダリングしようと思ったのですが、
いくつか解決の難しい問題があり、微妙に異なっていて申し訳ありません。
背景IBL画像は一致させてあります。

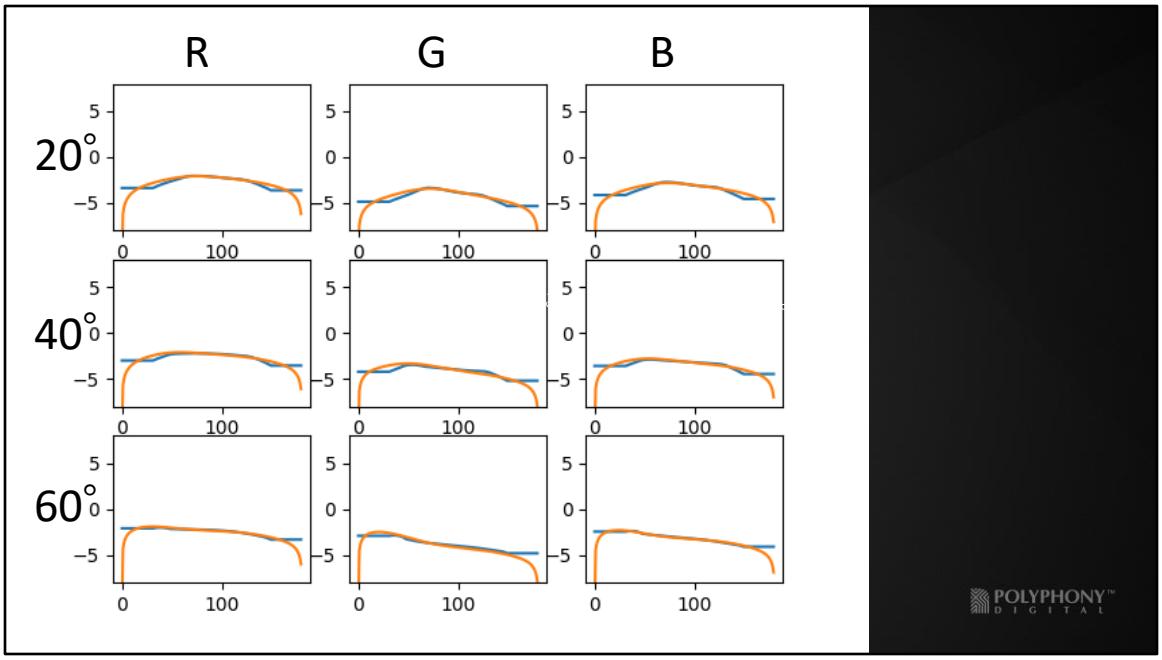


計測BRDFの断面と、フィッティングしたBRDFをプロットした図はこのようになります。
上から順に入射光20度、40度、60度
左から順に、R,G、Bとなっています。
青が計測BRDFで、オレンジが測色復元したPrincipledBRDFです。
特徴的なローブの広がりが難しく、少し再現に難があったようです。
Pythonにはたくさんの可視化エコシステムがありますので、
こうしてグラフで可視化することによって、今後の精度向上がしやすくなりました。

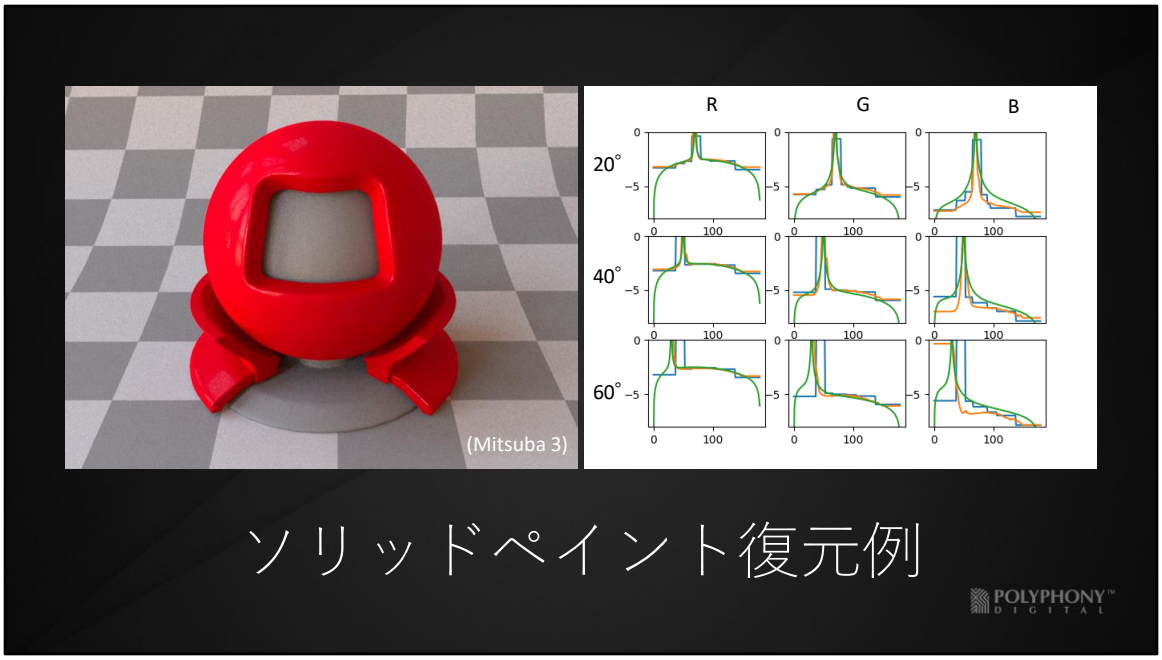
RGL EPFLとの比較

acrylic_felt_pink	新システム	旧システム
 (CG)	(CG)	 (CG)
Metallic	0.17855437099933624	0.86925488710403442
Roughness	0.6273486614227295	0.74151664972305298
Clearcoat	0	0.33530408143997192
Basecolor	185/19/99	102/0/56

マットなマテリアルに対しての復元結果です。
旧システムはなぜかメタリックかつクリアコートがあると主張しているのがわかるでしょうか？
ベースカラーも局所解に引きずられたのか、ズレてしまっていることが分かります。



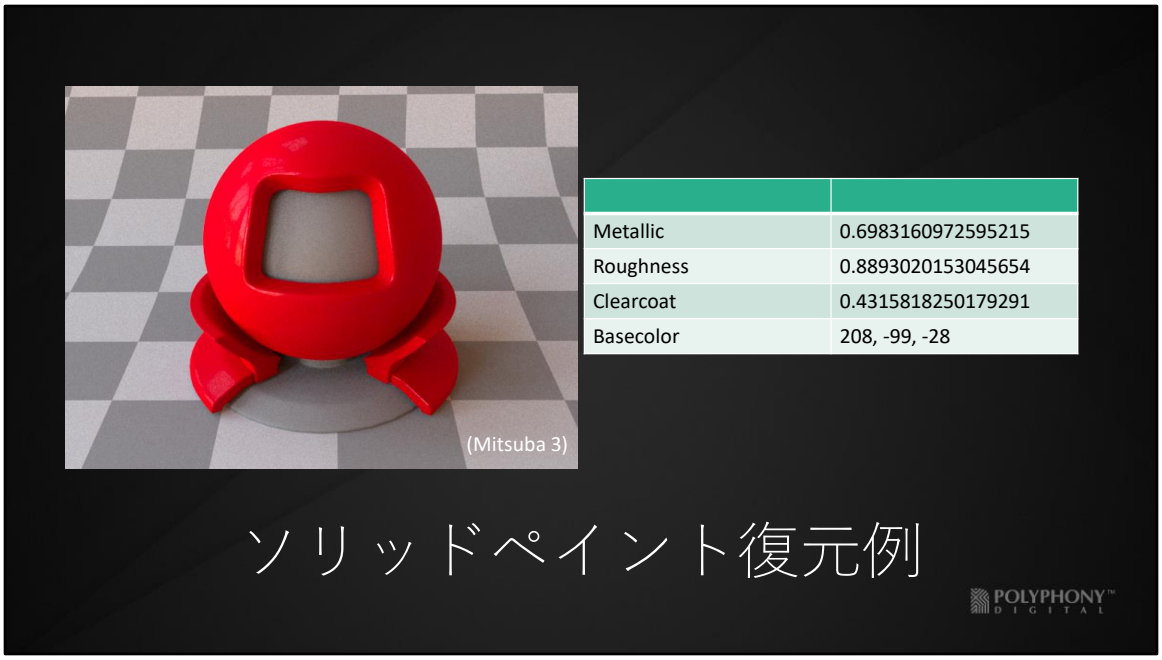
グラフで見るとかなり一致しているように見えますが、レンダリングしてみるとまだまだ差異があるように見えますので、
人間の感覚とは恐ろしい精度だなと思わされます。



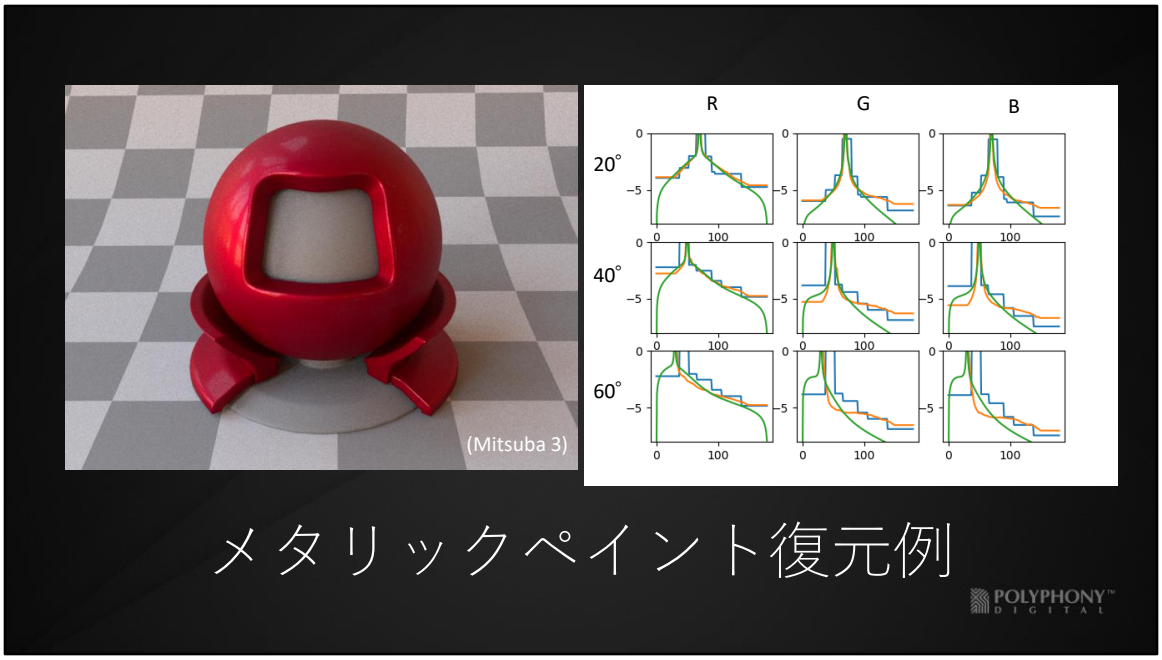
実際の塗装片からの計測データのフィッティング結果もいくつかご紹介します。
これはソリッドペイントの例で、単色の赤色ペイントです。
二つの計測器のデータを使っているため、グラフが3本あります。

緑色がフィッティング結果です。
おおむね形状が復元できていることが分かります。

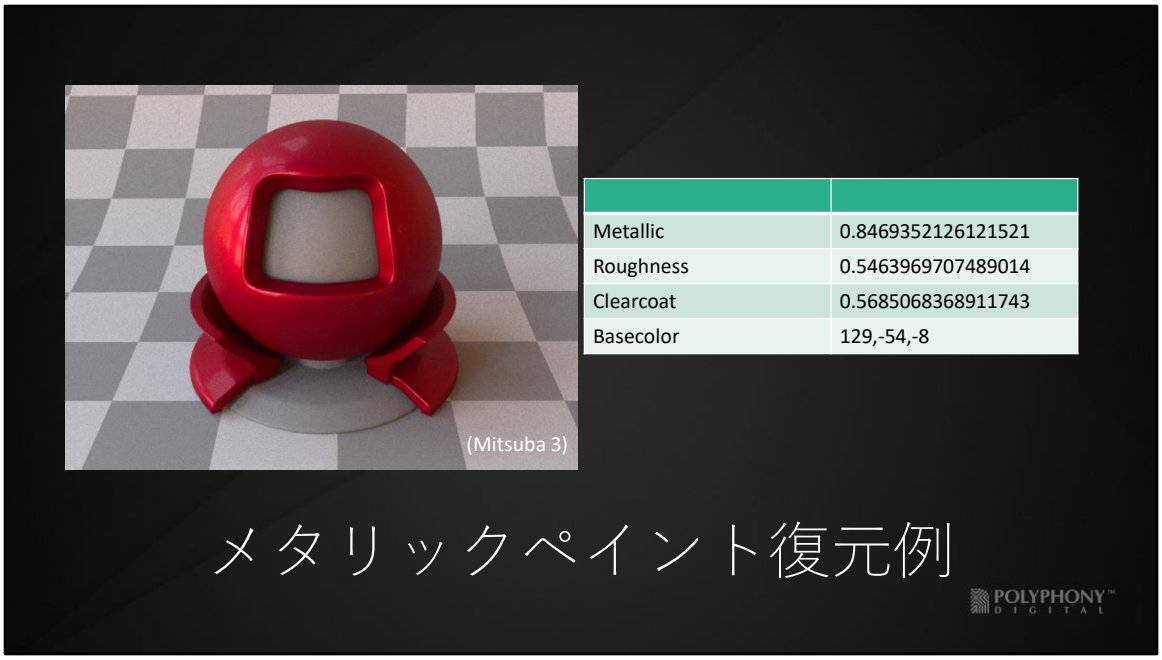
青チャンネルはもう少し頑張ってフィットできても良さそうですが、
一部の値が色域外になっているため、このようになっています。



復元結果のパラメータはこのようになりました。
この微妙な数値が、まさに数値最適化ならでは、という感じがします。
ベースカラーはRec709の色域外ですので、負の値になっています。

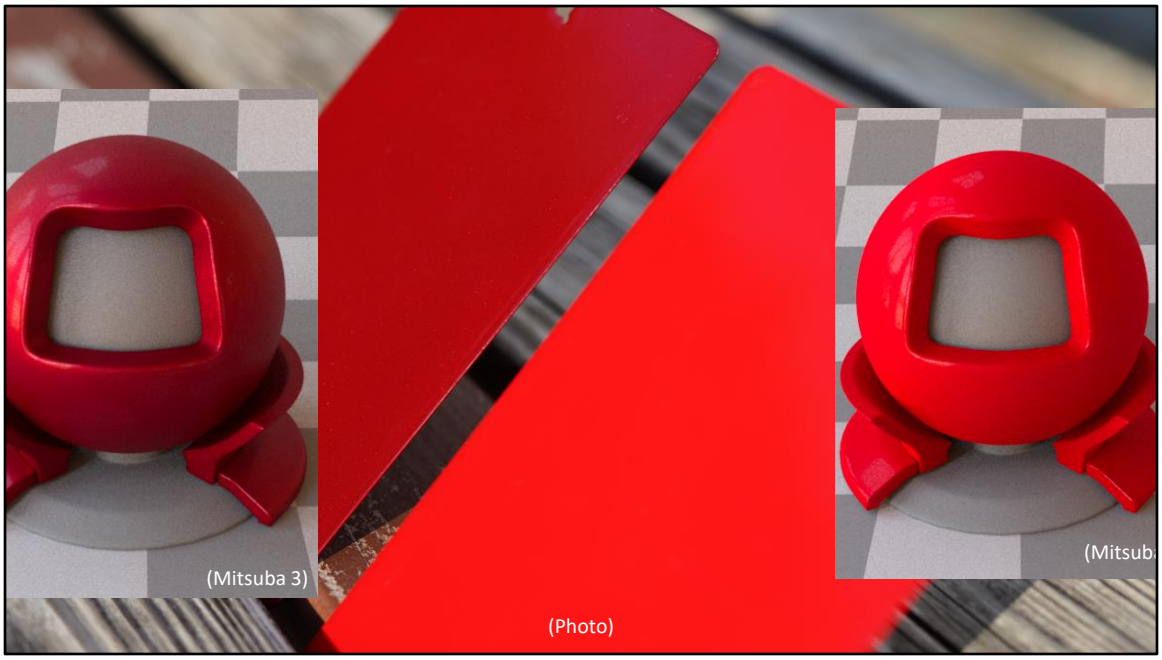


同じメーカーのメタリック塗装の例です。
素材感の差がきちんと再現できているのがわかるかと思います。



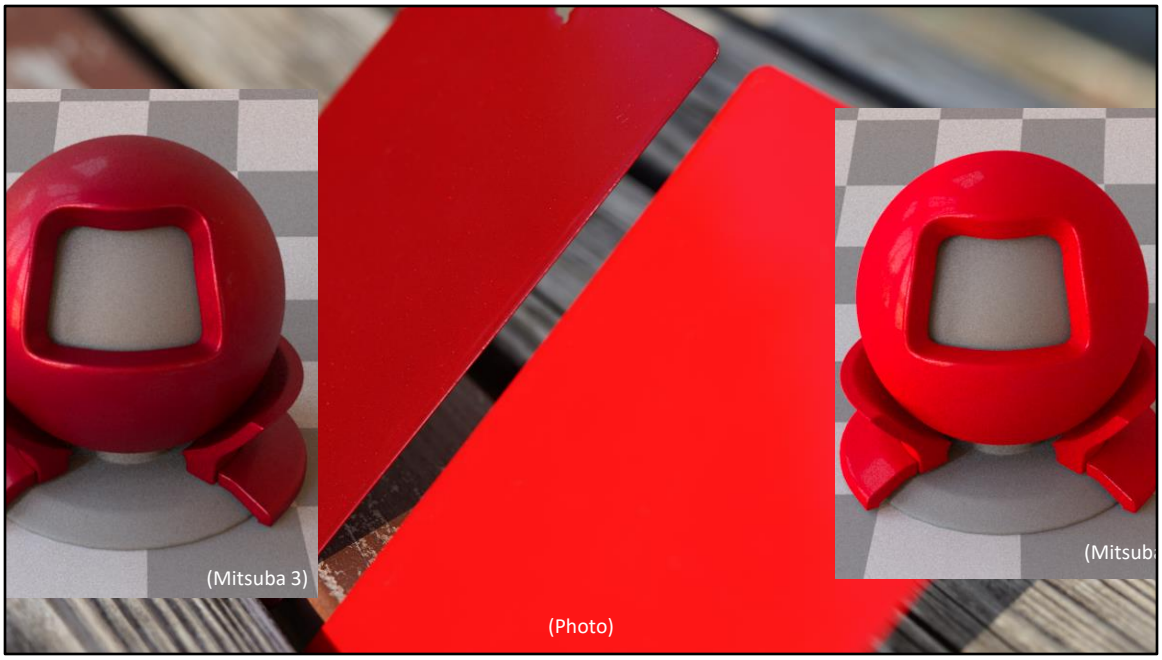
Metallic	0.8469352126121521
Roughness	0.5463969707489014
Clearcoat	0.5685068368911743
Basecolor	129,-54,-8

同じく復元結果のパラメータです。
よりラフネスが下がり、クリアコートが上がりました。



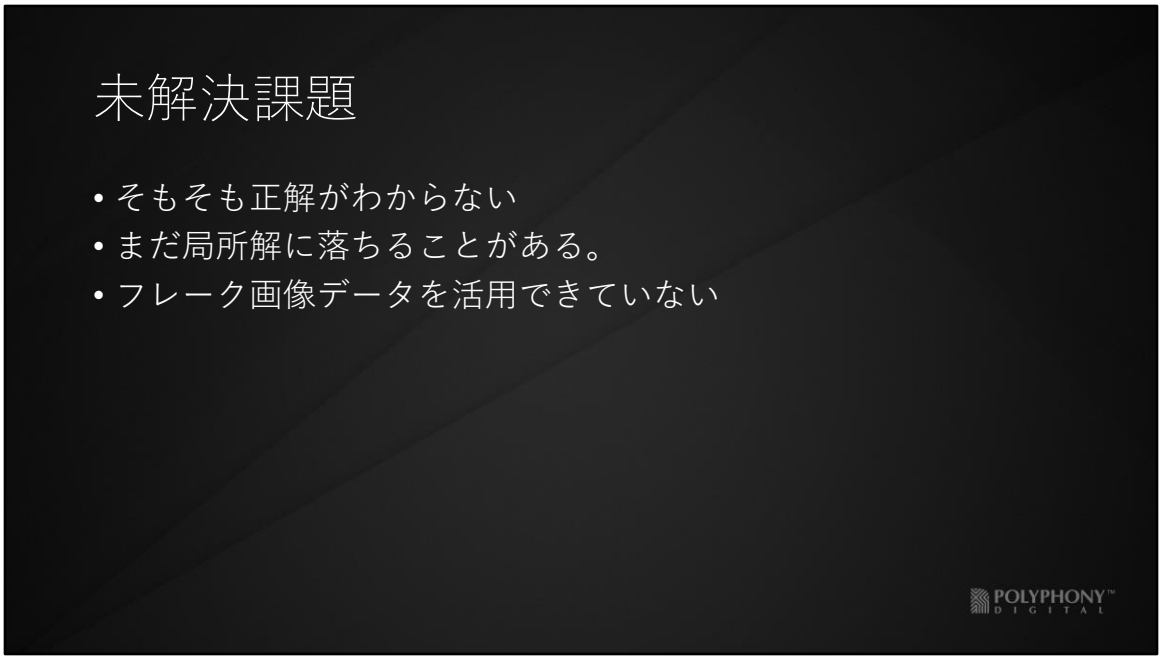
この実際の塗装片の写真と、その復元結果を並べるとこのようになります。
メタリック塗装はメタリックに、ソリッド塗装はソリッドに復元されている
のがわかるでしょうか。

映像でもお見せします。



それぞれ、印象はかなり良く復元できているのがお分かりいただけるでしょうか。
以上が、今回新しく開発した測色復元システムの、ご紹介でした。

おわりに

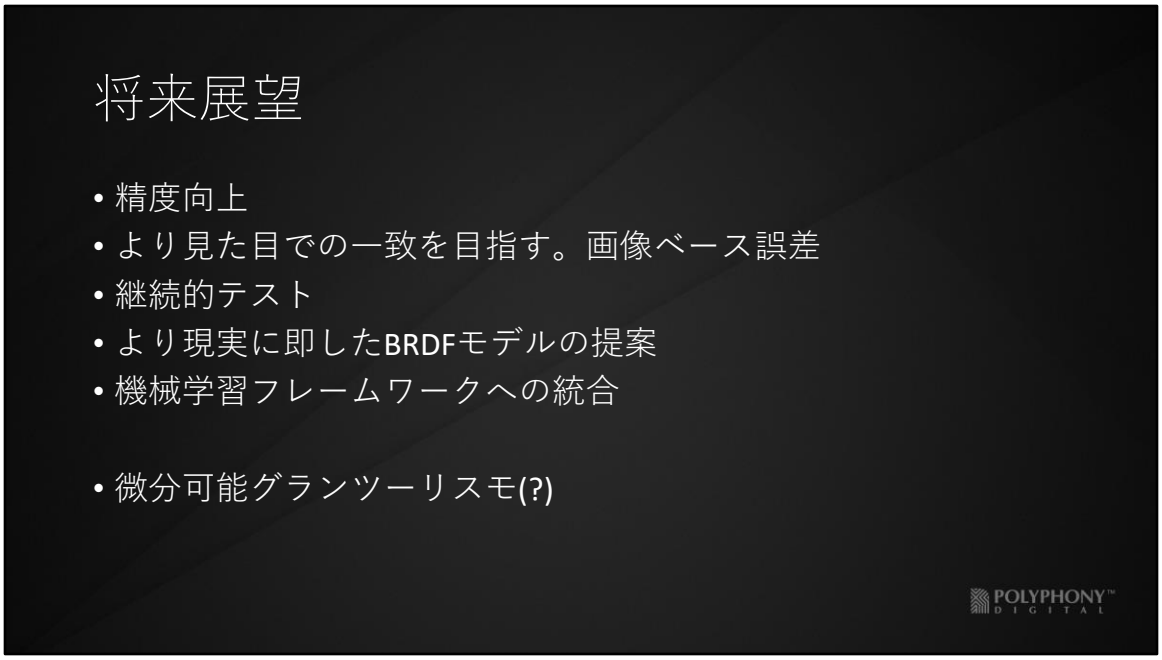


まだいくつかの問題が未解決として残っています。
一番の問題としては、正解がわからない、という点です。

マテリアルのパラメータを計測、公開している例はあまり聞いたことが無いため、
どのマテリアルがどのパラメータになれば正解なのかがわからない、という課題があります。

どうしても局所解に陥りやすいマテリアルがあり、過剰にツヤツヤになってしまったりする例があります。
BRDFモデルを別々に最適化するなどして、もっとロバストな最適化を行いたい。

今回の測色機には、画像センサが入っていますが、
このデータは今のところ活用できていません。
せっかくの微分可能レンダラですので、画像データも今後の最適化に利用していきたいと考えています。



今後の展望ですが、まず、再現精度を向上していきます。
次に、ロスをレンダリング画像における見た目の差などから求めるようなシステムに移行していきたいと考えています。
微分可能レンダラを用いているのはこの予定があるからです。
より定量的なテストを組み、システム更新時にデグレを防ぎたいと考えています。
今回得られたシステムで計測可能になったBRDF測定データをもとに、
今後はより現実に即したBRDFモデルなどを提案することも可能かなと考えています。
また、全段が微分可能になっていますので、機械学習フレームワークへの組み込みも考えられます。
大量の取材データから三次元構成を行うと同時にBRDFフィッティングするといったことも将来的には可能かもしれませんし、
Nerfや生成モデルとの相性も良いはずです。様々なアイデアを試していきたいと思っています。
自動微分や数値最適化、微分可能プログラミングはより大きな潮流となっていくはずです。
本公演が、みなさまのご理解の一助となりましたら幸いです。

謝辞(敬称,順序略)

- 石山 浩平
- 大垣 真二
- 鈴木 健太郎
- 平井 健太朗
- 安田 廉
- 高野 修一
- 山口 崇
- 佐藤利音
- ポリフォニー・デジタルみなさま

references

- [Burley2012] https://www.disneyanimation.com/publications/physically-based-shading-at-disney/
- [Wikipedia:BRDF] https://en.wikipedia.org/wiki/Bidirectional_reflectance_distribution_function
- [DRDC2017] https://cradpdf.drdc-rddc.gc.ca/PDFS/unc287/p805823_A1b.pdf
- [Dupuy2018] https://rgl.epfl.ch/publications/Dupuy2018Adaptive
- [Nicodemus1965] https://ui.adsabs.harvard.edu/abs/1965ApOpt...4..767N
- [XRite:MAT12] https://www.xrite.com/ja-jp/categories/portable-spectrophotometers/ma-family/ma-t12
- [Synopsys:Minidiff] https://www.synopsys.com/optical-solutions/scattering-measurements.html
- [Rusinkiewicz1998] https://www.cs.princeton.edu/~smr/papers/brdf_change_of_variables/
- [Misumi:Arasa] https://jp.misumi-ec.com/vona2/detail/223000850998/
- [Zeltner2021] https://tizianzeltner.com/projects/Zeltner2021Light/
- [Fukuda2010] https://cir.nii.ac.jp/crid/1520853834440072704
- [Miika2015] https://reality.cs.ucl.ac.uk/projects/two-shot-svbrdf/two-shot-svbrdf.html
- [Higo2008] https://repository.dl.itc.u-tokyo.ac.jp/record/1884/files/48066435.pdf
- [Desmos:Deriv] https://www.desmos.com/calculator/wtywqxysqw
- [github:micrograd] https://github.com/karpathy/micrograd
- [Cheung2004] https://onlinelibrary.wiley.com/doi/10.1111/j.1478-4408.2004.tb00201.x
- [Blender:Principled] https://docs.blender.org/manual/ja/2.90/render/shader_nodes/shader/principled.html
- [pytorch] https://pytorch.org/
- [autodiff] https://autodiff.github.io/
- [Loubet2019] https://rgl.epfl.ch/publications/Loubet2019Reparameterizing
- [Mitsuba3] https://www.mitsuba-renderer.org/
- [Kato2022] https://speakerdeck.com/hkato/wei-fen-ke-neng-rendaringu-cvimyan-jiu-hui-tiyutoriaru
- [Wenzel2022] http://rgl.epfl.ch/publications/Jakob2022Dr.jit
- [Tiancheng2018] https://cseweb.ucsd.edu/~ravir/tisbrdf.pdf
- [Sato2022] https://cir.nii.ac.jp/crid/1050294949472584832
- [XRite:xDNA] https://www.xrite.com/-/media/xrite/files/literature/l10/l10-300_l10-399/l10-365_introduction_to_xdna/l10-366_intro_to_xdna_en.pdf
- [RGL:MaterialDB] http://rgl.epfl.ch/materials
- [MERLBRDF] https://www.merl.com/research/license/BRDF
- [ColourScience] https://www.colour-science.org/
- [MazdaTR2017] https://www2.mazda.com/ja/technology/gihou/2017/wbbook/pageindices/index94.html#page=95
- [Takano2021] https://arxiv.org/abs/2106.03157
- [Wikipedia:AD] https://ja.wikipedia.org/wiki/%E8%87%AA%E5%8B%95%E5%BE%AE%E5%88%86

POLYPHONY™ DIGITAL